Appendix B

Web Service Choreography Interface (WSCI) 1.0

W3C Note 8 August 2002

This version:
http://www.w3.org/TR/2002/NOTE-wsci-20020808
Latest version:
http://www.w3.org/TR/wsci
Editors:
Assaf Arkin, Intalio arkin@intalio.com
Sid Askary, Intalio sid_askary@yahoo.com
Scott Fordin, Sun Microsystems Scott.Fordin@Sun.COM
Wolfgang Jekeli, SAP wolfgang.jekeli@sap.com
Kohsuke Kawaguchi, Sun Microsystems kohsuke.kawaguchi@eng.sun.com
David Orchard, BEA Systems dorchard@bea.com
Stefano Pogliani, Sun Microsystems stefano.pogliani@sun.com
Karsten Riemer, Sun Microsystems Karsten.Riemer@Sun.COM
Susan Struble, Sun Microsystems susan.struble@eng.sun.com
Pal Takacsi-Nagy, BEA Systems pal.takacsi@bea.com
Ivana Trickovic, SAP ivana.trickovic@sap.com
Sinisa Zimek, SAP sinisa.zimek@sap.com

Abstract

The Web Service Choreography Interface (WSCI) is an XML-based interface description language that describes the flow of messages exchanged by a Web Service participating in choreographed interactions with other services.

WSCI describes the dynamic interface of the Web Service participating in a given message exchange by means of reusing the operations defined for a static interface. WSCI works in conjunction with the Web Service Description Language (WSDL), the basis for the W3C Web Services Description Working Group; it can,

also, work with another service definition language that exhibits the same characteristics as WSDL.

WSCI describes the observable behavior of a Web Service. This is expressed in terms of temporal and logical dependencies among the exchanged messages, featuring sequencing rules, correlation, exception handling, and transactions. WSCI also describes the collective message exchange among interacting Web Services, thus providing a global, message-oriented view of the interactions.

WSCI does not address the definition and the implementation of the internal processes that actually drive the message exchange. Rather, the goal of WSCI is to describe the observable behavior of a Web Service by means of a message-flow oriented interface. This description enables developers, architects and tools to describe and compose a global view of the dynamic of the message exchange by understanding the interactions with the web service.

Status of This Document

This document is a submission to the World Wide Web Consortium (see Submission Request, W3C Staff Comment). This document is a NOTE made available by the W3C for discussion only. Publication of this Note by W3C indicates no endorsement of its content by W3C, nor that W3C has, is, or will be allocating any resources to the issues addressed by the Note. This document is a work in progress and may be updated, replaced, or rendered obsolete by other documents at any time.

This draft proposes a language standard that can be used in conjunction with existing Web-service protocols to provide a description of the observable behavior of Web services. It is provided by the authors for consideration and review by the web services community.

Feedback and comments are welcome and may be sent to BEA Systems (wsci@bea.com), Intalio (wsci@intalio.com), SAP AG (wsci-feedback@sap.com), or Sun Microsystems (wsci-feedback@sun.com).

Please note that feedback to any one of these aliases is automatically shared amongst all co-editors. For more information on the shared privacy policy regarding any feedback information, please visit any of the co-editor's sites where the WSCI specification is now hosted.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Table of Contents

---

1. Introduction

1.1 Overview

WSCI describes how Web Service operations – such as those defined by WSDL [WSDL]– can be choreographed in the context of a message exchange in which the Web Service participates. Interactions between services – either in a business context or not – always follow and implement choreographed message exchanges (processes). WSCI is the first step towards enabling the mapping of services as components realizing those processes.

WSCI also describes how the choreography of these operations should expose relevant information, such as message correlation, exception handling, transaction description and dynamic participation capabilities.

WSCI does not assume that Web Services are from different companies, as in business-to-business; it can be used equally well to describe interfaces of components that represent internal organizational units or other applications within the enterprise. Again, WSCI does not address the definition of the process driving the message exchange or the definition of the internal behavior of each Web Service.

WSCI describes the interdependencies among the Web Service's operations so that any client:

- Can understand how to interact with such service in the context of the given process; and
- Can "anticipate" the expected behavior of such service at any point in the process' lifecycle.

Being able to describe the dynamic interface of a service in the context of a particular process enables the developer/architect to abstract from the implementation and to focus on the role the Web Service plays in such process.

1.2 The Reference Framework

Web Services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web Service is an autonomous, well-defined, standards-based component that can be accessed via established Web-based protocols.

A "stack" of layered standards is emerging that aims to ensure semantic and technical interoperability of Web Services. This stack, developed by the W3C, is still in its early stages and is currently being built from the ground up; several additional layers are needed in order to enable true Web Service collaborations.

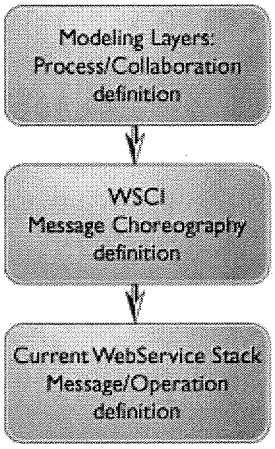


Figure 1-1

In parallel, other standards are building semantics and interoperability for business processes and collaborations in a top-down approach.

It is anticipated that these two stacks will meet in the middle. Although there is still a need for an overall architecture to make this happen in an effective way, WSCI provides the first step to linking the two.

WSCI is primarily a participant in the bottom-up stack, but it anticipates the emergence and integration of higher-level layers in the area of collaboration.

1.3 Problem

1.3.1 The current Web Services stack

The current Web Service stack does not define a choreography language or any other relationship among atomic operations. It does, however, provide the basic layers:

- SOAP defines the basic formatting of a message and the basic delivery options. A SOAP compliant Web Service knows how to send and receive SOAP-based messages
- WSDL describes the static interface of a Web Service. It defines the protocol and the message characteristics of end points by means of four basic operations; a WSDL compliant Web Service knows how to support any of these four operations. The operations defined in this way:
    - Are *atomic* in nature (that is, no intermediate state is visible from the outside)
    - Describe the *direction* of messages (incoming or outgoing) but not the

behavior of the service as a result of each individual operation.

The current Web Service technologies may be adequate for simple information retrieval in a stateless message exchange, such as a stock quote Web Service; for some services, it may even be sufficient for messages to maintain state in remote procedure calls by passing tokens or state variables within objects.

1.3.2 The need for describing complex interactions

Most Web Services, however:

- Participate in longer conversations, spanning beyond the boundaries of a single operation. In these conversations, the ability to perform certain operations depends upon the previous exchange of messages as well as the way to interpret the content of each message may be influenced by the previous exchange.

- Provide compound services, rather than atomic actions. For instance, this is the case of Web Services that interact with complex back-end business processes. Such processes are often automated and require a defined choreography of messages to properly operate and provide the service that is required.

Therefore, there is a need to address questions such as the following:

- *Can messages be sent and/or received in any order?*
  Without a clean and precise description of the message choreography supported by the Web Service, it would be difficult for a client – which wants to interact with the service – to properly understand the sequence of messages the Web Service is expecting and the sequence of messages it will deliver.

- *What rules govern sequencing of messages?*
  Without a precise description of the way in which information carried by the exchanged messages affect the observable behavior of a Web Service, it would be difficult for a client to properly manage situations in which messages of different types can be expected depending upon the value of some condition.

- *Is there any relation among any incoming and/or outgoing messages?*
  In order to properly prepare and interpret the messages that are exchanged within a given process, it is very important to understand and describe the correlation between these messages, i.e. the mechanism by which the Web Service can manage concurrent conversations.

- *Is there a "start" and an "end" of a given sequence? Can a given sequence*

be partially "undone"?
In order for a Web Service to be part of a long message exchange, it is important to clearly define when the message exchange is actually triggered, when it is considered to terminate and which parts of it can be considered as managed in a transactional way.

- *Can a global view of the overall exchange of messages be drawn?*
  A static Web Service definition language alone does not address the need for composing a global model of how two (or more) Web Services interact. Without something like WSCI, it is not possible to understand how the services interoperate and thus:
  - To verify if the service behaved as stated by a given process specification
  - To derive the choreography of the overall process to describe and thus, potentially, to modify it.
  - To monitor the behavior of the service on respect to all other participants in the message exchange

1.4 Challenges

The answers to questions such as the ones listed in the previous section are addressed by EAI middleware in a traditional, intra-enterprise environment,. But the loosely coupled, distributed nature of the Web prevents a central authority (or a centralized implementation of a middleware) from exhaustively and fully coordinating and monitoring the activities of the enterprise applications that expose the Web Services actually participating in the message exchanges. The reasons are:

- Traditional workflow models rely on message-based computing methods that are tightly coupled to protocols for business-to-business or application integration. These protocols assume a different, more tightly linked and controllable environment, which is not the nature of the Web.
- Participants in traditional workflow are, normally, known in advance. When moving to the Web environment, it is very likely that Web Services will be dynamically chosen to fulfill certain roles.
- Traditional models normally call for centralized engines, while the nature of the Web is decentralized.

This provides a fundamental challenge in defining and running a complex, process-aware Web Service based on traditional workflow models.

Thus, the challenge for WSCI is to provide answers to the previous (and other related) questions in the domain of the Web based computing. Fundamentally, the challenge will be to describe Web Services:

- From an external point of view (without knowing how internally they operate)
- Independently of any particular integration model (workflow, MOM etc)
- Precisely enough to allow other components to have a clear understanding of how to properly interact with them.
- In the context of each specific message exchange in which they participate.

1.5 Solution

To be part of a useful and manageable Web Service collaboration, individual operations must be allowed to convey enough information about how they can be used in a given scenario in order to enable them to participate in more complex processes.

WSCI achieves this by defining a layer on top of the existing Web Service stack. This layer describes the required behavior of a Web Service relative to the message exchange it must support.

WSCI supports the following key requirements for a long lasting, choreographed, stateful message exchange:

- **Message choreography:** a WSCI interface describes the order in which messages can be sent or received in a given message exchange, the rules which govern such ordering, the boundaries of a message exchange (when it starts and when it ends).

  WSCI does not define how the Web Service internally manages such choreography. Instead, it allows a Web Service to assert what is the observable choreography of messages it is capable of managing in the context of a given message exchange.

- **Transaction boundaries and compensation:** A WSCI interface describes which operations are performed in a transactional way and, thus, informs other participants of the capability of such Web Service to perform these operations in an "all or nothing" way. This capability also enables the Web Service to eventually join a distributed transaction with other services with which it interacts.

  WSCI does not define a two-phase commit protocol over the Web or how transactions are carried over by Web Services. Rather, it allows a Web

Service to assert when it is capable of managing operations in a transactional way, to precisely define the transaction boundaries and the externally observable compensation behavior.

- **Exception handling:** A WSCI interface describes how the Web Service will react when exceptional conditions happen, thus providing a description of alternative patterns of behavior.

  However, WSCI itself does not define the mechanism by which a Web Service reacts and manages exceptional situations and faults. Again, it allows a Web Service to describe its observable behavior when it's dealing with deviations from the normal behavior.

- **Thread management:** A WSCI interface describes if and how a Web Service is capable of managing multiple conversations (based on the same message exchange) with the same partner or with different partners. In the same way, it also describes the required relationship among parts of different messages belonging to the same message exchange.

  WSCI does not define how the Web Service is capable of managing multiple conversations in a concurrent way. It allows a Web Service to assert the observable elements of the exchanged messages that allow it to properly treat each conversation separately from the others. Further, WSCI allows a Web Service to describe the relationships among the parts of different messages that collectively guarantee the consistency of the message exchange.

- **Properties and Selectors:** A WSCI interface describes the elements that influence the observable behavior of a Web Service, such as alternatives based on the runtime values of some parts of the messages.

  Although WSCI is not an executable language that provides the semantic for variables used to automate it, it allows a Web Service to assert which are the relevant elements of the exchanged messages that influence its observable behavior at any given time.

- **Connectors:** A WSCI interface describes how the operations performed by different Web Services acting in the same message exchange actually link together. WSCI enables the mapping of "consume" operations from a Web Service to "produce" operations from another Web Service in order to unambiguously build a model of the global exchange. By means of the WSCI Global Model it is also possible to describe how interfaces from different services participating in the same message exchange can be linked together to build the end-to-end model of interactions.

  WSCI does not define the middleware connecting the Web Services involved

in the message exchange, nor does it define the order in which different Web Services happen in the course of a message exchange.

- **Operational context:** A WSCI interface describes how the same Web Service behaves in the context of different message exchanges.

  Different WSCI interfaces (observable behaviors) can be associated with the different operational contexts in which the Web Service participates. WSCI does not define the behavior of a Web Service independently on how it is used.

- **Dynamic participation:** A WSCI interface describes how the identity of the target service is dynamically selected. This selection is based on some criteria that are known at runtime and that depend on information described by the WSCI interface itself, such as message parts. By knowing the properties used to identify the target service, it is possible to understand how, changing the value of such properties, the behavior of the corresponding service is affected.

  The definition of a particular mechanism used to identify the target service is out of the scope of WSCI and can be addressed by existing and future specifications.

In summary, a WSCI interface describes all the artifacts required to provide:

- The external, observable view of how the Web Service interacts with other services within a given message exchange, or, in other words, how the Web Service is perceived to behave by the external world in the context of a given message exchange,

- The view of the message exchange as seen by the Web Service, or, in other words, how the Web Service perceives the behavior of the external world in the context of a given message exchange.

In doing this, the interface promotes the capabilities the Web Service provides in a given message exchange (transactional, correlation etc).

The above features and concepts are described in further detail in subsequent sections of this specification.

1.6 External Architecture

WSCI defines a new layer in the emerging stack of standards associated to Web Services.

WSCI works on top of the current Web Service stack and below layers in the

emerging Web Service architectural model that may be thought of as process or collaboration modeling layers.

WSCI describes the interface between an implementation and the message exchange (collaboration) in which it participates.

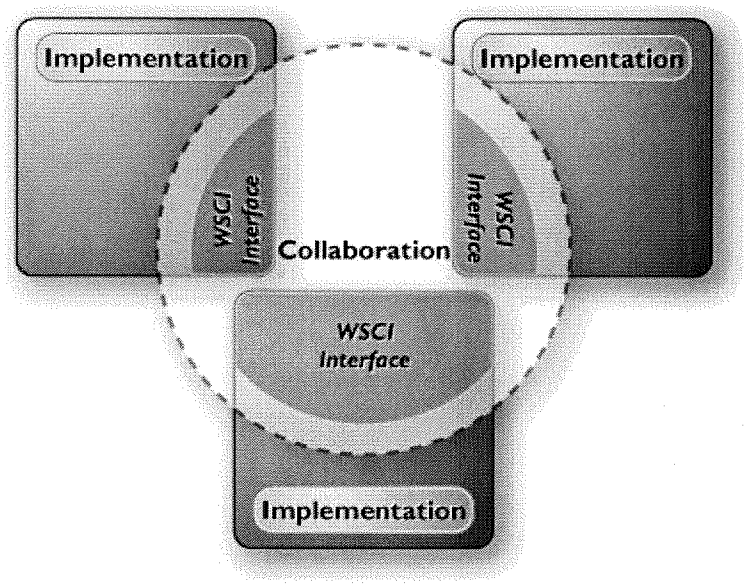


Figure 1-2

1.6.1 Relationship to Web Service Description Language (WSDL)

WSCI is designed to work in conjunction with Web Service description languages, which provide the static interface of a Web Service. While WSCI can work with different description languages, particular attention is devoted to the binding to WSDL, which is the basis for the W3C Web Services Description Working Group and rapidly gaining momentum.

Both WSDL and WSCI can be seen as interfaces. WSDL describes a static interface that evenly lists the entry points to the service. WSCI describes the dynamic interface by providing for the interrelationship between multiple operations in the context of a well-defined message exchanges. The very idea of WSCI as an interface lies in the fact that it can operate on WSDL (or any other static description language) artifacts, which are mainly messages exchanged by services.

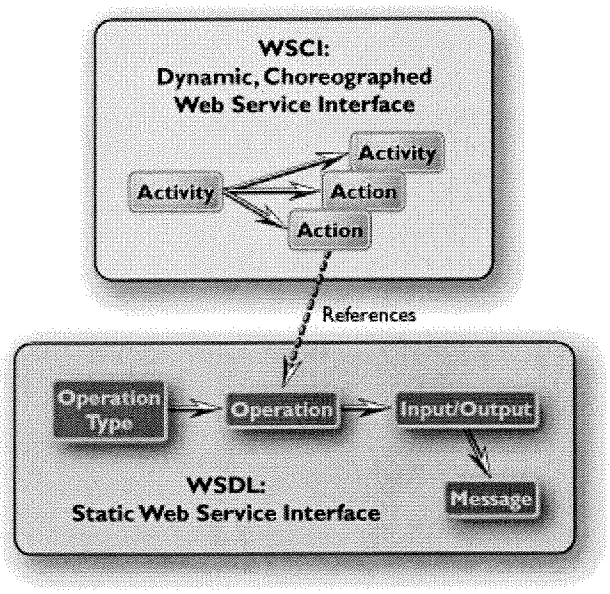


Figure1-3

When used in conjunction with WSDL, WSCI enables the description of complex choreographies across multiple operations as defined by WSDL. The messages and their associated "operations", "services", "ports" are directly referenced from the relevant WSDL definitions. Where WSDL describes each "operation" in a vacuum, WSCI enables the description of complex choreographies across multiple "operations". One of the basic concepts of WSCI – Action – directly maps to the execution of any of the four types of WSDL operations.

Whilst it is possible for a static description language to provide a single interface for a Web Service, it is very likely that the same Web Service exhibits more than one WSCI interface; each WSCI interface describes the observable behavior of that service in a specific message exchange context. It is also possible that the behavior in the same message exchange context would be described by multiple WSCI interfaces, each one providing a "view" for a different party.

1.6.2 Relationship to Implementations

WSCI is an interface description language. It describes the observable behavior of a service and the rules for interacting with the service from the outside. It does not specify the possible implementation on the inside. It is declarative and cannot, by itself, be executed. However, it is precise and unambiguous enough that external actors will know at each stage in the given process which messages that service may or must send or receive next.

The observable behavior of each party in a message exchange is described independently of the others; each party could actually be implemented by completely abstracting from its WSCI description; WSCI's main goal is to describe

an observable behavior, not to define how the implementation works.

Thus, in a message exchange, any Web Service described by WSCI can interact with:

- Other Web Services, whose implementation has been derived by their WSCI description
- 'Hard-coded' software components with internally encoded mechanisms to guarantee the correct sequence of the exchange
- Or human controlled software agents where the human determines the sequence of interaction within the constraints of the WSCI description.

For that matter, the message exchange could actually be several human controlled software agents interacting.

Although WSCI has no explicit constructs for implementation, it is assumed that every sending of a message is mapped to some identifiable processing in the implementation (actually, the processing required for or resulting in the sending of the message). Likewise, it is assumed that every receipt of a message is mapped to some identifiable processing in the implementation (actually, the processing triggered by the receipt of the message).

Non-observable mappings, such as the use of some internal software are outside the scope of WSCI.

A specific kind of mapping between messages and implementation that may be observable is the recursive use of services, so that an incoming message might cause the invocation of another lower level or auxiliary service through a separate message exchange.

1.6.3 Relationship to Collaborations

Business processes are increasingly relying upon collaboration. Businesses must automate these collaborative processes in order to achieve greater productivity. Likewise, government agencies, educational institutions, and non-profit organizations are looking to Web-based collaboration to increase efficiency and expand horizons.

The ultimate purpose for a Web Service is to facilitate such collaborations; in order for a Web Service to fully represent the role of a participant in these collaborations and to properly grant the required level of interoperability, many layers of capabilities need to be addressed.

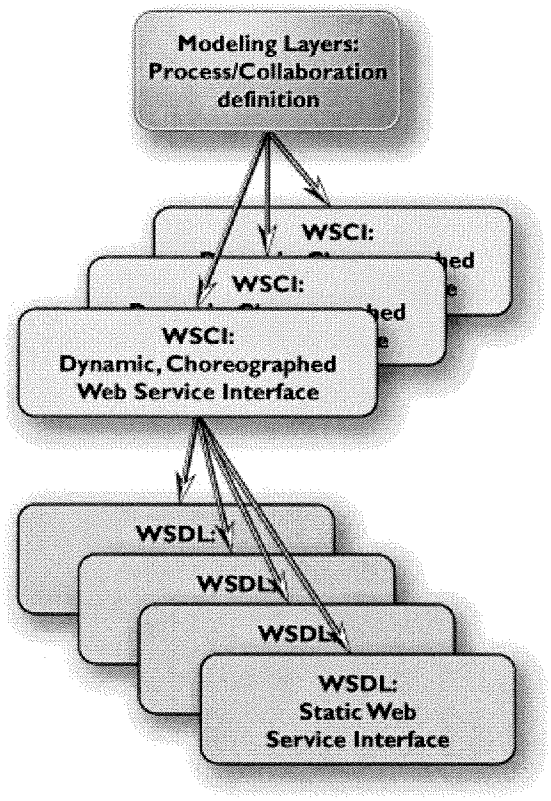


Figure 1-4

Whilst a process or collaboration modeling language may support full definition of roles, responsibilities, contracts, artifacts, state management and state transitions, the goal of WSCI is to present the interface that a given Web Service exhibits in such a complex scenario, when it is called to cover one or more roles.

Thus, when used in a collaboration context, the objective of WSCI is to unambiguously describe the choreography of message exchanges that involve a specific Web Service covering one (or more) role(s) in such collaboration. Each Web Service is considered a peer in the collaboration and, as such, exhibits its own interface as part of the collaboration.

1.6.4 Relationship to Workflow

WSCI is not a "workflow description language"; it is envisaged that this role will be covered by some other specification that would properly address the description of collaborative processes.

WSCI can describe the observable behavior of a Web Service interacting with a workflow; as well, it can describe the observable behavior of a system that implements a workflow (or which behaves as such).

WSCI does not address the definition of the behavior required by a Web Service when it needs to coordinate its own activities with the activities performed by other Web Services; but it can describe the observable behavior of a Web Service behaving as such.

Thus, in a model describing the way in which multiple participants interact, WSCI addresses the description of the "boundaries" for each participant.

1.7 Simple Example

The example presented in Section 5, taken from the well-known Travel Reservation System use case, describes the use of most of the WSCI features in the context of a real-life example.

In order to familiarize the reader with key WSCI artifacts, a subset of the full example is presented here. This subset does not account for all the modeling possibilities offered by WSCI but is simple enough to show its power. Section 3 provides a more thorough description of how additional WSCI features can be used to better model the overall scenario and to arrive to the full-fledged model described in Section 5's example.

1.7.1 Scope

The following example illustrates, in the scope of a real life use case, how WSCI models the following concepts:

- The Action, the basic construct of WSCI, and its binding to some WSDL operation
- How multiple actions are choreographed in a simple sequence
- How groups of actions can be grouped in a process which represents an identifiable (and reusable) behavior exhibited by the Web Service
- The use of Correlation to describe how messages exchanged within a choreography are related
- How to define the WSCI interface of a Web Service

1.7.2 Simple scenario

The scenario describes the basic behavior that is exposed by a Web Service implementing the functionalities of a Travel Agent in a Travel Reservation System; this scenario will be enhanced in Section 5 to better reflect the real-life use case and to show the use of the majority of the WSCI language features.

In this scenario, the Travel Agent only interacts with a Traveler according to a very simplified choreography:

1. A Traveler can order a trip with the Travel Agent
2. Some time later, the Traveler confirms the previous order
3. The Travel Agent bills the Traveler for the trip

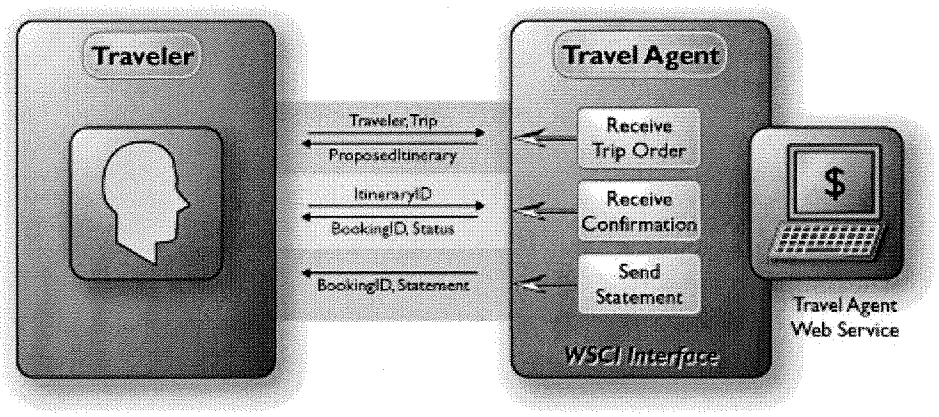


Figure 1-5

The goal is to show how WSCI enhances the model provided by WSDL.

1.7.3 The static definition

Here is a representation of the Travel Agent Web Service as provided by a static definition language, such as WSDL.

```
<? xml version = "1.0" ?>
<definitions name = "Travel Agent Static Interface"
   targetNamespace = "http://example.com/consumer/TravelAgent"
   xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema"
   xmlns:tns = "http://example.com/consumer/TravelAgent"
   xmlns = "http://schemas.xmlsoap.org/wsdl/">

   <types>
     <schema xmlns = "http://www.w3.org/2000/10/XMLSchema">

<!-- ************************************************************* -->
<!-- ***************** COMPLEX TYPES ********************** -->
<!-- ************************************************************* -->
        <complexType name = "Traveler">
        <sequence>
          <element name = "name" type = "xsd:string" />
          <element name = "travelerID" type = "xsd:string" />
        </sequence>
        </complexType>
```

```
        <complexType name = "trip">
        <sequence>
          <element name = "itineraryID" type = "xsd:string" />
          <element name = "startDate" type = "date" />
          <element name = "startCity" type = "xsd:string" />
          <element name = "arrivalDate" type = "date" />
          <element name = "destinationAirport"
               type = "xsd:string"/>
          <element name = "numberOfSeats"
               type = "nonNegativeInteger"/>
          <element name = "preferredCarrier"
               type = "xsd:string" />
          <element name = "comments" type = "xsd:string" />
        </sequence>
        </complexType>

        <complexType name = "proposedItinerary">
        <sequence>
          <element name = "itineraryID" type = "xsd:string" />
          <element name = "startTime" type = "timeInstant" />
          <element name = "startAirport" type = "xsd:string" />
          <element name = "destinationTime"
               type = "timeInstant" />
          <element name = "destinationAirport"
               type = "xsd:string" />
          <element name = "carrier" type = "xsd:string" />
          <element name = "availability" type = "boolean" />
          <element name = "totalCost" type = "float" />
          <element name = "validityDeadline"
               type = "timeInstant" />
        </sequence>
        </complexType>

        <complexType name = "statement">
        <sequence>
          <element name = "bookingID" type = "xsd:string" />
          <element name = "creditCard" type = "tns:CCInfo" />
          <element name = "date" type = "date" />
          <element name = "amount" type = "float"/>
          <element name = "transactionID" type = "xsd:string"/>
        </sequence>
        </complexType>

        <complexType name = "CCInfo">
        <sequence>
          <element name = "number" type = "xsd:string" />
          <element name = "issuer" type = "xsd:string" />
          <element name = "expiryDate" type = "month" />
        </sequence>
        </complexType>
    </schema>
  </types>

<!-- ************************************************************ -->
<!-- ****************** MESSAGES ******************************** -->
```

```
<!-- ************************************************************ -->
   <message name = "tripOrderRequest">
     <part name = "traveler" type = "tns:traveler"/>
     <part name = "trip" type = "tns:trip"/>
   </message>

   <message name = "tripOrderAcknowledgement">
     <part name = "proposedItinerary" type = "tns:proposedItinerary"/>
   </message>

   <message name = "bookingRequest">
     <part   name = "itineraryID" type = "xsd:string"/>
   </message>

   <message name = "bookingConfirmation">
     <part   name = "bookingID" type = "xsd:string"/>
     <part   name = "status" type = "xsd:string"/>
   </message>

   <message name = "statement">
     <part   name = "bookingID" type = "xsd:string"/>
     <part   name = "body" type = "tns:statement"/>
   </message>

<-- ************************************************************ -->
<-- *************TRAVEL AGENT PORT TYPES ************************ -->
<-- ************************************************************ -->
   <portType name = "TAtoTraveler">
     <documentation>
        This port type references the operations the Travel Agent
        performs with the Traveler service
     </documentation>

     <operation name = "OrderTrip">
        <input message = "tns:tripOrderRequest"/>
        <output message = "tns:tripOrderAcknowledgement"/>
     </operation>

     <operation name = "bookTickets">
        <input message = "tns:bookingRequest"/>
        <output message = "tns:bookingConfirmation"/>
     </operation>

     <operation name = "SendStatement">
        <output message = "tns:statement"/>
     </operation>
   </portType>
</definitions>
```

1.7.4 What is missing

From the previous example, it is clear that a static interface, whilst able to capture important information about the atomic operations that are performed by the Travel Agent Web Service, is not enough to cope with:

- **Choreography aspects.**
  The WSDL definition does not describe how to interpret a set of operations happening in a given order; in this example, it is not clear if all the 3 operations supported by the Travel Agent Web Service should happen (and in which order) in order for a Traveler to complete the reservation of a trip.

- **Correlation aspects.**
  Even if exposing the basic choreography of the supported operations, it is not possible to describe which information allow the Travel Agent Web Service to associate a confirmation request with a previously submitted trip request.

Other important behavioral elements that are introduced in the following sections include:

- **Transactions.**
  The WSDL definitions of the Travel Agent Web Service does not describe if the Web Service operations are performed in a transactional way. It may be important for the Traveler to know that a trip request would be logically rolled back in case of any error happening during the process.

- **Possible Choices.**
  In real life, the Travel Agent Web Service may be able to choose the execution of different actions based on some information that can be carried in the message exchange:

  - The Web Service may be able to accept a Confirmation request or a Cancellation request and it will react accordingly. In this case the choice is based on the type of the incoming message.

  - The Web Service may be able to send either a Bill or a Notification that the trip cannot be confirmed based on the validity of the Credit Card info provided by the Traveler or based on the availability of the trip as communicated by the Airline Reservation system.

1.7.5 The WSCI definition

The following code example shows how the previous WSDL static interface can be enriched using WSCI concepts.

```
<? xml version = "1.0" ?>
<wsdl:definitions name = "Travel Agent Dynamic Interface"
   targetNamespace = "http://example.com/consumer/TravelAgent"
   xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema"
   xmlns:tns = "http://example.com/consumer/TravelAgent"
   xmlns = "http://www.w3.org/2002/07/wsci10">                    1.
```

```
<!-- WSDL complex types -->                                          2.
<!-- WSDL message definitions -->
<!-- WSDL operations and port types -->
<!-- selectors -->

<correlation name = "itineraryCorrelation"                          3.
        property = "tns:itineraryID">
</correlation>

<interface name = "TravelAgent">                                    4.
  <process name = "PlanAndBookTrip"                                 5.
       instantiation = "message">                                   6.

     <sequence>                                                     7.
     <action name = "ReceiveTripOrder"                              8.
         role = "tns:TravelAgent"                                   9.
         operation = "tns:TAtoTraveler/OrderTrip">                 10.
     </action>

     <action name = "ReceiveConfirmation"                          11.
         role = "tns:TravelAgent"
         operation = "tns:TAtoTraveler/bookTickets">
        <correlate correlation="tns:itineraryCorrelation"/>        12.

       <call process = "tns:BookSeats" />                          13.
     </action>

     <action name = "SendStatement"                                14.
         role = "tns:TravelAgent"
         operation = "tns:TAtoTraveler/SendStatement"/>
     </action>
     </sequence>
  </process>

  <process name = "BookSeats" instantiation = "other">             15.
     <action name = "bookSeats"
        role = "tns:TravelAgent"
        operation = "tns:TAtoAirline/bookSeats">
     </action>
  </process>
</interface>
</wsdl:definitions>
```

The important information that the WSCI interface provides on top of the static WSDL interface:

1. The WSCI interface definition is embedded inside the *wsdl:definitions* element; it does not require a root element nor does it require to be written in a separate file. The reason for this choice is explained in Section 3.1.2.

2. The standard WSDL definitions for types, messages, operations and *portTypes* as seen in the previous code example (see Section 1.7.3) apply.

3. The *itineraryCorrelation* definition indicates that messages carrying the same *itineraryID* refer to the same trip; this will help the implementation of the Web Service to properly manage concurrent executions of the same message exchange (from the same or different users).

4. The *interface* element is the WSCI container; it is designed to contain all the WSCI process definitions that describe the dynamic behavior of the Web Service in the context of a given message exchange (in our case, the Travel Reservation System process).

   Each WSCI interface has a name to properly distinguish multiple interfaces exposed by the same Web Service.

5. The *PlanAndBookTrip* process models the observable behavior of the Travel Agent Web Service in the context of the Travel Reservation System process; thus it describes how the Travel Agent Web Service behaves in that context.

   The *PlanAndBookTrip* is the image of the Travel Reservation System process as seen by the Travel Agent Web Service.

   The *process* element is, in WSCI, the basic unit of re-use; thus an *interface* can only contain process definitions.

6. The process is qualified with the "`instantiation=message`" attribute. This means that the execution of the process will be triggered as soon as the first action referred to by the process is ready. In this case, the *PlanAndBookTrip* process will be triggered when the *tripOrderRequest* message is received by the Travel Agent Web Service (this is the consume message referenced by the *OrderTrip* operation).

7. The *sequence* construct indicates that the 3 WSCI actions are executed sequentially.

   According to this, the Web Service highlights that the 3 WSDL operations associated with the 3 WSCI actions cannot happen just in any order but that the Travel Agent Web Service can engage in the Travel Reservation System process (*PlanAndBookTrip*) only by performing the 3 actions in the correct order.

8. The *action* is the basic WSCI construct; it identifies a unit of work associated with a given operation.

   Here, the WSCI interface describes that the Travel Agent executes the *OrderTrip* operation whilst performing the *ReceiveTripOrder* action.

9. A WSCI interface can describe the observable behavior of a Web Service in

the context of a given message exchange. The Web Service can represent more than one logical role in such exchange; the "`role=travelAgent`" attribute specifies that the *ReceiveTripOrder* action is executed on behalf of the *travelAgent* role.

10. The operation attribute associates the WSCI action with a WSDL operation as specified by the WSDL definitions. Via this operation, it will be possible to determine the type of the operation (1-way or 2-way), the messages that are actually exchanged and the bindings.

11. The *ReceiveConfirmation* is the second action in the sequence; after accepting a trip order from the Traveler, the Travel Agent Web Service is able to accept a confirmation for the booking.

    This means that if the Travel Agent Web Service receives another *tripOrderRequest* message, a new instance of the same process is created instead than continuing the current one.

12. It is important to be able to correlate the *ReceiveConfirmation* action with the *ReceiveTripOrder* action that could have happened long time before; this is done via the reference to the *itineraryCorrelation* correlation.

    In this example, if the value of the *itineraryID* field carried by the *BookingRequest* message is the same as the *itineraryID* field carried by the previous *tripOrderRequest* message, then the *ReceiveConfirmation* action is part of an already established conversation (the one that was previously started when the *ReceiveTripOrder* action was executed).

    The *itineraryCorrelation* is implicitly instantiated when the *ReceiveTripOrder* action is executed.

13. The *BookSeats* process is called as part of the WSCI action. This means that the actions defined by the *BookSeats* process are executed between the consume and the produce parts of the *bookTickets* operation.

    The use of the *call* construct allows re-use of process definitions.

14. The *SendStatement* action does not require a Correlation since it references a produce operation.

15. The *BookSeats* process is defined with "`instantiation=other`". This means that this process will never be instantiated by the reception of a triggering message but can only be instantiated when explicitly invoked.

    This process is not described in detail since it references the relations between the Travel Agent and the Airline Reservation system.

1.7.6 Benefits

The simple WSCI example shown before highlights how WSCI can help the Web Service architect and programmer.

- The architect of the Travel Agent Web Service can provide the WSCI definition of the service, along with the static definition. The WSCI definition is precise enough to describe the behavior of the Travel Agent Web Service when used inside the Ticket Reservation System process.

  This may become part of a standard worksheet associated with the Web Service.

- Anyone interested in interacting with the Travel Agent Web Service knows its expected behavior. A programmer that is interested in building a software agent for the Traveler can use the Travel Agent Web Service WSCI definition to properly build it. The WSCI definition allows the client programmer to clearly understand how to interact with the Travel Agent Web Service.

- The WSCI interface could also be of help for the programmer of the Travel Agent Web Service itself.

  Sometimes WSCI can be used to describe the observable behavior of an existing Web Service; sometimes it can be used to define how a newly created Web Service should behave. In the latter case, the server programmer could use the WSCI definition to build the communication layer on the top of some legacy code that implements the basic functionalities of the Travel Agent service.

1.8 Notational Conventions

1.8.1 Keywords

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [RFC 2119].

1.8.2 Namespaces

The following namespace prefixes are used throughout this document:

Prefix	Namespace URI	Definition
wsdl	http://schemas.xmlsoap.org/wsdl/	WSDL namespace for WSDL framework.

xsd	http://www.w3.org/2001/XMLSchema	Schema namespace as defined by XSD.
wsci	http://www.w3.org/2002/07/wsci10	WSCI namespace as defined by this document.
tns	(various)	The "this namespace" (`tns`) prefix is used as a convention to refer to the current document.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with "http://example.com" represent some application-dependent or context-dependent URI [RFC 2396].

1.8.3 Informal Syntax

The definition of each WSCI element is given in XML-like grammar using the `monospace` typeface. The definition of an element is shown with the element name enclosed in angle brackets.

Required attributes appear in **bold** typeface. Where the attribute type has an enumerated type definition, the values are shown as separated by vertical bars. Where the attribute type is given by a simple type definition, the type definition name from either XSD or the WSCI schema is used. Where the attribute is optional and has a default value, it is shown following a colon.

Support for extension attributes is shown by {extension attribute}. Where used in the grammar, it indicates support for any number of attributes defined in a namespace other than the WSCI namespace.

The allowed content of WSCI elements is shown using a simple grammar:

- An element name is used for any content part that must be an element of that type
- A name enclosed in curly braces and appearing in italic typeface refers to content parts of some other type. For example, {*any activity*} refers to any element that defines an activity.
- Support for extension elements is shown by {extension element}. Where used in the grammar, the content part may be any element defined in a namespace other than the WSCI namespace.
- Support for mixed content is shown by {mixed}. Where used in the grammar,

the allowed content is a mix of character data and of elements defined in any namespace.

- The cardinality of any content part is specified using the operators "?" (zero or one), "*" (zero or more) and "+" (one or more). If no operator is used, the content part must appear exactly once. Cardinality that cannot be expressed using either operator is shown using curly braces, with the minimum and maximum values separated by comma, e.g. "{2,*}" denotes two or more.

- Content parts may be grouped together using parentheses "( )" to form a new content part. A choice group consists of all consecutive content parts, separated by a vertical line "|". A sequence group consists of all consecutive content parts that are separated by a comma ",".

XSD schemas are provided as a formal definition of WSCI grammar (see WSCI Schema, Section 8.4).

---

2. Language Overview

This section introduces the main modeling concepts underlying the WSCI interface and the WSCI model definitions as well as the extensibility features of the WSCI language.

2.1 Concepts

The following concepts are part of the underlying model:

- Interface
- Activities and choreography of activities
- Processes and units of reuse
- Properties
- Context
- Message correlation
- Exceptions
- Transactions and compensation activities
- Global model

2.1.1 Interface

WSCI aims to describe how Web Services participate in choreographed, long-lasting and stateful message exchanges. More precisely, WSCI aims to describe the externally observable behavior of a Web Service in such a message exchange. The focus of the described behavior is on the temporal and logical dependencies among the messages the Web Service exchanges with one or more

other services in the context of a given scenario. WSCI maps this description to the notion of an *interface*.

The details of the behavior of the Web Service are described in the processes that are contained in the interface. Usually, an interface contains at least one process whose execution can be initiated by the receipt of a message. This indicates that the service will not proactively perform the behavior described in that process, but instead will wait until triggered by the respective message.

A Web Service may expose multiple interfaces for supporting multiple scenarios. In this sense, each interface describes how a Web Service is perceived to behave in the context of a particular scenario. A Web Service may also expose multiple interfaces for supporting different views of the behavior of the service in a single scenario. Such a view might, for instance, show only the interactions of the service with one specific client, thus allowing the client to fully understand how to interact with the service in that scenario.

2.1.2 Activities and their choreography

WSCI describes the behavior of a Web service in terms of choreographed activities. Activities may be atomic or complex, i.e. recursively composed of other activities. Choreography describes temporal and/or logical dependencies among activities.

Atomic activities represent the basic unit of behavior of a Web service, such as sending and/or receiving a message, or waiting for a specified amount of time. They are termed atomic because any observer of the Web service cannot observe intermediate states during their execution. Atomic activities dealing with messages correspond to the execution of operations defined in static service definition languages such as WSDL.

Complex activities are recursively composed of other activities; ultimately, each complex activity is composed of actions. Each complex activity defines a specific kind of choreography for the activities it is composed of. WSCI supports the definition of the following kinds of choreographies:

- *Sequential execution*: The activities must be executed in sequential order.
- *Parallel execution*: All activities must be executed, but they may be executed in any order.
- *Looping*: The activities are repeatedly executed based on the evaluation of a condition or an expression. WSCI supports for-each, while, and repeat-until style loops.
- *Conditional execution*: One out of several sets of activities is executed based

on the evaluation of conditions (`switch`) or based on the occurrence of an event (`choice`).

2.1.3 Processes

A process is a portion of behavior that is labeled with a name. The behavior described by a process can be reused by referencing its name. As such, the process is the WSCI unit of reuse.

The way in which a process is used (or re-used) is by properly instantiating it. A process can be instantiated by:

- Receipt of one or more messages that are defined as triggering the choreography described by the process.
- Calling or spawning the process (see below). The respective Call or Spawn statements are explicitly shown in the interface.
- Instantiating the process from within the service implementation. In this case, the actual instantiation of the process is not shown in the interface.

In each of the above cases, the Web Service needs to be in a state where instantiating the process is allowed. Whether or not a process can be instantiated also differs on the type of the process. WSCI allows the definition of two types of processes:

- *Top-level processes:* These processes are defined at the interface level and can be referenced from everywhere within the interface.
- *Nested processes:* These processes are defined within complex activities. A nested process can be referenced only from within the complex activity that defines it.

A process can be referenced using a Call or a Spawn statement:

- *The process is called:* The behavior contained in the process will be executed, while the call waits for the called process to complete. This is very similar to calling a subroutine in a programming language.
- *The process is spawned:* The behavior contained in the process will be executed in a parallel thread of control. The parallel thread can later be joined again.

The ability to call processes introduces the concept of reuse into WSCI. The ability to spawn/join processes introduces the concept of parallel execution of activities into WSCI.

2.1.4 Properties

Properties are introduced in WSCI as a modeling artifact used to reference a "value" within the interface definition. They are the equivalent of variables in other languages. The concept of properties, though, provides an additional layer of abstraction in WSCI. This layer eliminates dependencies between the interface definition and abstract message definitions (such as those defined by WSDL). Additionally, it allows referencing values instantiated by the service implementation.

Properties can be used to avoid, within the interface definition, explicit references to abstract messages (or message parts) described in a service definition language (such as WSDL). This is accomplished in two ways:

- For each incoming or outgoing message there is a property with the same value as the message content and with the same name as the abstract message name. The property is instantiated upon the receipt of the incoming message or the construction of the outgoing message, respectively. It has the same data structure as the abstract message.

- A property may holda value obtained applying an expression to a part of an incoming message. The value of that property is obtained from any incoming message that includes the particular message part. The service should behave as if the value of the property is derived every time the service receives a message containing the specific message part. WSCI introduces the `selector` element (see Section 3.2 for more details), which defines how the value of a property is obtained from an incoming message.

Properties may also be used to reference a value that is instantiated by the service implementation and could be used in the interface definition. Such properties are useful for specifying abstract conditions for loops and branches in message choreographies.

Accordingly, properties can be used as:

- An abstraction of the message itself;
- A reference to a value established from a message part upon receipt of a message;
- A reference to a value established by the service itself upon instantiation.

A property is defined as a name-value pair. The name must be unique among all properties defined in the same scope. Properties do not need to be explicitly defined in order to be used. Consequently, their values are not explicitly restricted to any data type. WSCI does not prescribe any type system and values of properties may be of any data type. Properties are visible in the context in which they are instantiated including all its parent contexts, unless they are defined as

local properties to that context.

Properties do not have to map to any concrete artifact implemented by the service; they are just a modeling artifact for WSCI and can be used whenever the description of the external behavior of the service requires referencing to "named objects". For this reason, a precise algebra for properties has not been specified.

2.1.5 Context

Context is a WSCI concept that describes the environment in which a set of activities is executed. Each context definition pertains to a particular set of activities, and each activity is defined in exactly one context definition. The context definition for top-level processes, when not present, is assumed to be empty. The context definition describes the environment for the set of activities it pertains to in terms of:

- The set of declarations that are available to the activities;
- The set of exceptional events that might occur during the execution of the activities, and the exceptional behavior triggered by those events;
- The transactional properties associated with the execution of the activities.

A context definition may contain two different kinds of declarations: local properties, and local process definitions.

- Local properties are only available to the activities executing in this context.
- Local process definitions designate exactly those processes that may be instantiated in this context. A local process can only be called or spawned from a context where the definition of the local process is visible (see the scoping rules for nested contexts below). Called/spawned processes execute in the context where they are defined.

Context definitions may be nested to an arbitrary level. The scoping rule for nested contexts specifies that contexts see "outward" but neither "inward" nor "sideward". This means that an activity can refer to all local declarations defined in its own context and, recursively, in all of its parent contexts; this is referred to as the *execution context* for that activity. An activity cannot refer to local declarations defined in other contexts. Local declarations override local declarations with the same name in parent contexts.

Web Services may implement the concept of context in many different ways; WSCI does not prescribe nor encourage any particular such implementation, but aims to describe in a neutral way how the presence of context affects the observable behavior of the Web Service.

2.1.6 Message correlation

In WSCI, a conversation represents a message exchange between two or more services participating in a particular scenario. A service can be engaged in multiple conversations at the same time, with the same or with different services. The concept of correlation describes how conversations are structured and which properties must be exchanged to retain the semantic consistency of the conversation. A correlation is not limited to a single conversation between two participants; it can span multiple conversations between different participants.

In the example presented in Section 5, a traveler exchanges messages with the travel agent. Initially the traveler will submit a chosen trip to the travel agent. The travel agent will, as response, return an itinerary. After that the traveler has a choice to change the proposed itinerary or make the reservation. This conversation between the traveler and the travel agent will be extended over a period of time.

The travel agent can enter into many conversations with the same traveler. For example, the traveler can send multiple requests for creating different itineraries for different trips. A request to change a proposed itinerary sent from the traveler to the travel agent need to be associated with a particular conversation – one that deals with the specific itinerary. For each new trip a new conversation is initiated. Each conversation is related to a particular trip request. For each trip request the travel agent will return a unique itinerary identification number, which the traveler must use in all subsequent messages related to the proposed itinerary (e.g. change the proposed itinerary or make the reservation).

In the example, we assume that the travel agent will contact the same airline company for each leg of the journey. Generally, it is possible that for different legs of the same journey a travel agent contacts different airline companies. In this case the travel agent engages in multiple conversations with different airline companies. These multiple conversations are considered as part of the larger conversation that relates to a single trip request. These (sub)conversations are distinguished by the itinerary identification number and the leg identification number.

Correlation is the mechanism by which a message received by the service is associated with a particular conversation. Different conversations can be distinguished by correlation instances. The correlation instance is a set of properties' values. Each message that belongs to a conversation must be designated by a particular correlation instance. The correlation properties are communicated as part of messages exchanged in the actions.

The service may implement the way in which it actually correlates messages belonging to the same logical conversations in multiple ways. WSCI aims to model this mechanism as seen from outside and it does not prescribe any implementation.

2.1.7 Exceptional behavior

WSCI allows declaring exceptional behavior that is exhibited by a Web Service at a given point in a choreography. The declaration of exceptional behavior is part of the context definition, and associates exceptions with a set of activities that the Web Service will perform in response to those exceptions. It is possible to declare the occurrence of the following kinds of exceptions:

- Receipt of a particular message that is considered as exceptional in that context.
- Occurrence of a fault; this fault might correspond to the receipt of a WSDL fault message, or to the generation of a fault by the service itself.
- Occurrence of a timeout.

The occurrence of an exception causes the current context to be terminated after the activities associated with the exception have been performed; at this point, the behavior defined in the parent context is resumed. Thus, WSCI supports the concept of "recoverable exceptions" that do not cause the overall choreography to terminate (similar to the throw/catch concept in Java). However, the occurrence of a fault for which no exceptional behavior is defined causes the context to be terminated and the fault to be raised in the parent context. More abstractly, exception handling behavior defined in a parent context definition can act as the default behavior for all its children, and in reverse, exception handling behavior defined in a context definition can be used to override exception handling behavior defined in its parent.

Exceptions are a WSCI modeling artifact designed to model the exceptional behavior exhibited by a Web Service at a given point of a conversation; they do not need to necessarily represent any "technical failure."

2.1.8 Transactional behavior

A context may be associated with a transaction. The transaction describes, from an interface perspective, the transactional properties of the activities that are executed in this context. Basically, the presence of transaction asserts that the set of activities is executed in an all-or-nothing manner, i.e. either the context terminates with no exception, in which case the transaction completes successfully, or the context terminates with an exception, in which case the effects of the activities executed so far are assumed to be rolled back by the private implementation of the service.

A transaction may declare a set of compensation activities that will be executed if the transaction has completed successfully, but needs to be undone. Note that the compensation, being a part of the service interface, describes only the externally

observable activities required to undo the transaction; compensation does not describe how, from an implementation point of view, the transaction is undone. For instance, assume that a travel agent has performed a transaction to book a hotel room for a traveler. If this booking has to be undone because there is no available flight to that location, the travel agent will probably send a notification of the un-booking to the traveler. In other words, the travel agent service describes its undoing of the "book hotel room" transaction by the compensation activity of sending the relevant notification to the traveler.

A transaction is either atomic or open-nested.

- **Atomic transactions** have no further inner structure. It is assumed that participants exchanging messages in the context of an atomic transaction rely on some kind of mechanism (e.g. CORBA OTS, JTS) to coordinate the outcome of the transaction, but specification of a specific such mechanism is outside the scope of WSCI. It is also outside the scope of WSCI to specify how the rollback of an atomic transaction is performed.

- **Open-nested transactions** are composed of other transactions, which in turn may be atomic or open-nested. If an open-nested transaction needs to be rolled back, the currently open sub transactions within the open-nested transaction are rolled back first (recursively), and after that the already completed sub transactions are compensated for in reverse order of completion.

2.1.9 Global Model

As described previously, a WSCI interface models the externally observable behavior of a particular Web Service in a choreographed, long-lasting and stateful message exchange with one or more other services; thus a WSCI interface describes the view of the overall message exchange as seen from one participant. WSCI allows also describing a multi-participant view of the overall message exchange by means of the WSCI Global Model.

The Global Model is described by a collection of interfaces of the participating services, and a collection of links between the operations of communicating services as described by a "static" service description language, such as WSDL. Links between operations indicate that the respective services will exchange messages across those links, i.e. there will be direct message flow(s) between those operations. Therefore, linked operations must be mirror images of each other.

In most scenarios, the information provided by the global model allows to derive between which actions the messages will actually flow. This allows visual modeling, analysis, validation and simulation of the overall message exchange.

The Global Model operates at the logical level of "Service Type". Since actions specify the role on behalf of which the service acts, and since each WSCI action is related to an operation, it is possible to infer the source and target roles for any action described by a WSCI interface from the operation links defined in the Global Model. This is particularly important for granting a top-down mapping from the description of the collaboration view of the overall process.

2.2 Extensibility

Two kinds of extensions of WSCI are allowed:

- Extensions introducing additional semantics that is not part of the normative WSCI specification, and
- Extensions that are used to expand the interface and the model definitions and that do not change the semantics defined in the normative WSCI specification.

There are three ways to accomplish the first type of extensions:

- By adding new types of activities (e.g. a *goto* construct); this is achieved by using the substitution group mechanism introduced in [XML Schema 1]. It is possible to define a new activity that uses `wsci:activity` as its substitution group (see Section 3.5.1 for more details about WSCI activity type). Substitution groups are not allowed in any other case.
- By extending the semantics of actions; this is achieved by means of the extensibility features for the action element (see Section 3.4.1). For instance, the WSCI specification describes a normative extension, which is used to identify the service against which the action is performed (see Section 6).
- By referencing a WSCI construct from within a document edited according to another specification; this is achieved by using strict naming rules that allow every process, activity, transaction and local property to be unambiguously referenced. For example, RDF can be used to annotate a WSCI interface definition with additional semantics.

The second type of extensions is accomplished supporting extensibility elements. They are used to allow other (existing or future) specifications to extend the interface definition and the model definition by adding elements and/or attributes defined in namespaces different from WSCI namespace. These extensions do not change the semantics defined in the normative WSCI specification; they are optional and must use an XML namespace different from that of WSCI. The use of extensibility elements is restricted to specific tags identified by the informal syntax used in Section 3 (the *{extension element}* and *{extension attribute}* placeholders).

An extensible WSCI element can be extended by:

- Zero or more attributes
- And/or at most one element (with the exception of the `action` element which may be extended by more than one element) defined in a namespace different from WSCI namespace.

The following table lists the extensible WSCI elements.

WSCI element	Extension attribute	Extension element
`<selector>`	Supports the definition of selectors that use data types in type systems other than the XML Schema type system.	Supports the definition of selectors that use expressions different from XPath expressions (e.g. XQuery).
`<action>`	Supports operations defined by a service definition language other than WSDL.	Supports definition of optional normative or non-normative semantics, e.g. locate services.
`<condition>`	Supports definition of conditions using a language other than XPath.	N/A
`<connect>`	Supports definition of operation connectivity when a service definition language other than WSDL is used.	Supports definition of advanced forms of operation connectivity, e.g. a definition providing the mapping of data types.
`<locator>`	Supports other information for the purpose of locating services.	Supports definition of locator mechanisms.

3. Language Elements

3.1 Introduction

This section describes the WSCI language in detail. We first introduce some principles and notions that apply to all or most of the language elements.

3.1.1 Terminology conventions

Most of the concepts that are introduced by WSCI (such as `process`, `context`, or `transaction`) have the following aspects:

- The actual WSCI syntax (the "syntactical" concept)
- The use of the syntax in an instance of the WSCI schema (the "design time" concept)
- An actual instance of the latter (the "run time" concept).

In order to distinguish these aspects precisely, a consistent terminology will be used in the remainder of this document. This terminology can be explained using the concept of process.

- The term "process construct" refers to the WSCI syntax.
- The term "process definition" refers to an occurrence of the `processelement` within a WSCI document.
- The term "process instance" refers to the conceptual instance of such a definition. The term "process" is used sometimes instead of "process instance."

3.1.2 The root element

The WSCI top-level definitions (the `interface`, `selector`, `correlation`, and `model` elements) are nested inside the `wsdl:definitions` element. This should not be understood as if WSCI extended WSDL, but as a reuse of the `wsdl:definitions` element due to the fact that this element is convenient and WSDL is the only specification that, as of today, defines a container with the expected semantic. WSCI does not define a `wsci:definitions` element, envisaging that a neutral container will be defined for all specifications acting in the Web service area.

Since WSCI reuses the `wsdl:definitions` and the `wsdl:import` elements, retaining their respective semantics, any tool needs to fully understand the syntax and semantics of these two WSDL elements in order to properly process WSCI.

3.1.3 Documentation and naming

Most of the WSCI constructs may be documented using the `documentation` element, and may be named using the `name` attribute. The semantics for these two syntax fragments will be explained here since it is the same wherever the fragments appear.

The `documentation` element is used to provide any kind of useful and human-readable documentation. Usually, the `documentation` element will contain pure text, but it may also contain elements defined in a foreign namespace. A WSCI processor will not interpret the contents of the documentation element.

The `name` attribute is required for correlation definitions, interface definitions, and the `model` element. It is optional for activity definitions, transaction definitions, fault definitions and the `connect` element. The value of the `name` attribute is a non-qualified name. It serves the purpose to reference the named entity from

elsewhere. For instance, correlation definitions are referenced in the `correlateelement`, process definitions are referenced in the `spawn` or `call` elements, and interface definitions are referenced from within the global model.

For each of the WSCI top-level constructs (interface, model, and correlation), the names of definitions of that construct form a scope of their own. This means, for instance, that the name of a correlation definition must be unique among the names of all correlation definitions within a given namespace, but need not be different from names of interface or model definitions in the same namespace. The same holds for names of interface definitions and model definitions.

Within a process definition, WSCI only enforces name uniqueness for all local property declarations and nested process definitions within the same context definition (see also Section 3.5.4).

3.2 Selector

Selector is a mechanism to abstract a complex message into a set of properties of interest to the service interface. Each abstract message may consist of one or more logical parts associated with a data type from some type system, e.g. the XML Schema type system [XML Schema 1], [XML Schema 2]. Assuming that a message part is associated with a particular data type allows decoupling between a property and the part of the message from which the value of such property is obtained. The selector definition references a data type from some type system and not an abstract message definition thus allowing the value of a property to be obtained from any message that includes the particular message type.

New message definitions can be added without affecting a previously defined selector. Also, the message structure can be changed (e.g. a part can be excluded); still the same selector definition can be used provided that the particular message part has not been changed. In this way selectors improve reuse and eliminate dependencies on abstract message definitions. They ensure that the interface definition is more resilient to changes. The same selector can be applied to any message that contains the message part associated with the same data type that is referenced in the selector definition. Selectors should be used when a property is derived from a message part using a non-trivial expressions (e.g. sum, count) or the property value can be obtained from any message that contains the message part associated with the particular data type.

The selector definition specifies how the value of a property is obtained from an incoming message. The service should behave as if the value of a property is derived every time the service receives a message containing the specific message part.

The syntax of the `selector` element is:

```
<selector property = QName
   element = QName
   type = QName
   xpath = expression
   {extension attribute}>
   Content: (documentation?,
   {extension element}?)
</selector>
```

The `property` attribute specifies which property is instantiated (or modified) by the selector. The property is identified by its fully qualified name to allow the referencing of properties defined in any namespace.

The `element` and `type` attributes are used to specify type references. The *element* attribute refers to an XML Schema element using a qualified name. The *type* attribute refers to an XML Schema simple type or complex type using a qualified name. These two attributes are mutually exclusive and optional. If both attributes are omitted an extension attribute must be used to reference a type system. The referenced type system must be different than the XML Schema type system. Exactly one type reference must be used.

> **Note:** XML Schema elements and types have different name scopes. It is valid to define an element with the same name as a complex type. Therefore two different attributes are defined in order to distinguish which definition is used.

The `xpath` attribute provides an XPath expression [XPath]that extracts a node-set from a message part in order to obtain the value for the property. This attribute is optional. If the `xpath` attribute is used, the content of the selector element may consist only of the documentation element. If the `xpath` attribute is omitted and the content of the selector element is empty or consists only of the documentation element, the complete message part is used to get the value for the named property. In this case the message part must be identified either by an XML Schema element or type definition.

The `selector` element is an extensible element. An extension attribute can be used to define a selector that operates on a type system other than XML Schema type system. Zero or more extension attributes may be specified. An extension element can be used to define a selector that uses expressions different from XPath expressions (e.g. XQuery expressions). The selector allows at most one extension element to be used.

If the `selector` element contains an extension element, the *xpath* attribute must be omitted. This extension declares how the value of the named property will be obtained from the message part. The message part may be identified either by an

element or type from the XML Schema type system or using any other data type.

A selector definition is not limited to a specific interface definition. The selector element is a WSCI top-level element.

Example. The value of the itinerary number property can be carried in different messages. Each of these messages has a message part of either simple type `itineraryID` or element type `itinerary`. Consequently, two different selectors must be defined to obtain the value of that property. The accompanying types are provided below.

```
<selector property = "tns:itineraryNo"
            type = "tns:itineraryID"
            xpath = "./text()" />

<selector property = "tns:itineraryNo"
            element = "tns:itinerary"
            xpath = "./itineraryID/text()" />

<xsd:simpleType name = "itineraryID">
   <xsd:restriction base = "xsd:string" />
</xsd:simpleType>

<xsd:element name = "itinerary" type = "tns:itineraryType" />

<xsd:complexType name = "itineraryType">
   <xsd:sequence>
      <xsd:element name = "itineraryID" type = "xsd:string" />
      <xsd:element name = "leg" type = "xsd:leg"
                   minOccurs = "1" maxOccurs = "unbounded" />
   </xsd:sequence>
</xsd:complexType>
```

The following selector definition does not specify any expression. The complete message part of element type `itinerary` is used to get the value for property `itinerary`. This property has type `itinerary`.

```
<selector property = "tns:itinerary"
      element = "tns:itinerary"/>
```

The value of property `legCount` is calculated from any message having a message part of element type `itinerary` using the following selector:

```
<selector property = "tns:legCount"
      element = "tns:itinerary"
      xpath = "count(./leg)" />
```

The value of property `legNo` is obtained from messages, which include a message part of element type `itinerary`. The property contains identification numbers of all legs of a given itinerary. Complex type `leg` used in the definition of complex type

`itineraryType` is provided below.

```
<selector property = "tns:legNo"
            element = "tns:itinerary"
            xpath = "./leg/legID" />

<xsd:complexType name = "leg">
   <xsd:sequence>
     <xsd:element name = "legID" type = "xsd:string" />
     <xsd:element name = "startTime"
                  type = "xsd:timeInstant" />
     <xsd:element name = "startAirport" type = "xsd:string" />
     <xsd:element name = "destinationTime"
                  type = "xsd:timeInstant" />
     <xsd:element name = "destinationAirport"
                  type = "xsd:string" />
     <xsd:element name = "carrier" type = "xsd:string" />
     <xsd:element name = "seat" type = "xsd:string" />
     <xsd:element name = "cost" type = "xsd:float" />
     <xsd:element name = "availability" type = "xsd:boolean" />
   </xsd:sequence>
</xsd:complexType>
```

3.3 Message Correlation

The correlation mechanism is a general mechanism used in WSCI to describe how a Web service distinguishes between different conversations in which it participates. An informal description of the mechanism and its usage is provided in Section 2.1.6.

WSCI defines two elements, `correlation` and `correlate`, that implement the correlation mechanism. The `correlation` element specifies a set of properties used for the purpose of defining the correlation identity. The `correlate` element (defined in Section 3.4.2) is used to associate an action with the correlation definition.

3.3.1 Correlation

The syntax of the `correlation` element is:

```
<correlation name = NCName
   property = list of QName
   extends = QName>
   Content: (documentation?)
</correlation>
```

The `name` attribute holds the name of the correlation. This name must be unique among names of correlations defined in the same namespace. It is used to reference the correlation from a correlate element. The `name` attribute is mandatory.

The `property` attribute lists all the properties forming the correlation identity and used to identify a correlation instance. At least one property must be specified and no property can be repeated twice in the same correlation definition. The properties can be listed in any order. A property is referenced using a qualified name.

A correlation can be defined as an extension of another correlation. In this case the correlation is a specialization of the extended correlation. It includes all properties that are part of the base (extended) correlation and adds one or more new properties. The base correlation is specified using the extends attribute. Additional properties are specified using the property attribute. The `extends` attribute is optional.

A correlation definition is not limited to a specific interface definition. The correlation element is a WSCI top-level definition.

Example. Two correlations are illustrated in this example: `itineraryCorrelation` and `legCorrelation`. The former correlation is based on the itinerary number property. The latter presents an extension of correlation `itineraryCorrelation` and is used to identify the conversation related to a specific leg of the itinerary. The `legCorrelation` correlation includes property `itineraryNo` that is part of the base correlation and adds one more property (i.e. the `legNo` property).

```
<correlation name = "itineraryCorrelation"
             property = "tns:itineraryNo" />
<correlation name = "legCorrelation"
             property = "tns:legNo"
             extends = "tns:itineraryCorrelation" />
```

3.4 Atomic behavior

3.4.1 Action

An action is an atomic activity that describes the manner in which the service uses an elementary operation in a context, in particular one involving the exchange of messages with other services.

```
<action
  name = NCName
  operation = QName/NCName
  role = QName
  {any attribute with non-WSCI namespace}>
  Content: (documentation?,correlate*,call?,
           {any element with non-WSCI namespace}*)
</action>
```

When the action performs an operation defined by WSDL it uses the `operation` attribute. This attribute names the WSDL operation that is performed by the action.

The WSDL definition is referenced using the fully qualified name of the port type definition and the non-qualified name of the operation definition:

```
@operation =: portTypeName'/'operationName
portTypeName =: QName
operationName =: NCName
```

For example, the value "`tns:traveler/findItinerary`" refers to the WSCI operation "`findItinerary`" that is part of the port type definition " traveler" that is defined in the same namespace as the WSCI interface.

An action can be associated with one of the following types of WSDL operations:

- **One-way:** The action performed by the service receives a message. A correlation should be used to associate the input message with the context in which the action occurs.
- **Request-response:** The action performed by the service receives a message and sends a response back to the sender. A correlation should be used to associate the input message with the context in which the action occurs.
- **Notification:** The action performed by the service sends a message to another service. No correlation is required.
- **Solicit-response:** The action performed by the service sends a message to another service and waits for a response.

The `role` attribute associates a role name with the action. The `role` attribute is optional and takes the form of a fully qualified role name. The `role` attribute can be used to reference a role definition that is given by some other specification using the role's qualified name. An interface can perform actions on behalf of multiple roles.

3.4.1.1 Example

- The Traveler service indicates its intention to order the trip by performing the solicit-response operation `OrderTrip` defined as part of the port type definition `tns:TravelerToTA` as follows:

```
<action name = "OrderTrip"
        role = "tns:Traveler"
      operation = "tns:TravelerToTA/OrderTrip"/>
```

- The Traveler service exposes the request-response operation *ReserveTickets* defined as part of the port type definition *tns:TravelerToTA* in the following way:

```
<action  name = "ReserveTickets"
         role = "tns:Traveler"
         operation = "tns:TravelerToTA/ReserveTickets">
<correlate correlation="defs:reservationCorrelation"
         instantiation= "true" />
</action>
```

- The following action references the request-response operation `ChangeItinerary` that the Travel Agent service exposes and that is defined as part of the port type definition `tns:TAtoTraveler`. Upon receipt of the request message, the service invokes a nested process for verifying seat availability (`tns:VerifySeats`), and waits for it to complete before returning the response message. The request message is correlated to the instance of this process through the `itineraryCorrelation`.

```
<action name = "ChangeItinerary"
        role = "tns:TravelAgent"
        operation = "tns:TAtoTraveler/ChangeItinerary">
  <correlate correlation="defs:itineraryCorrelation"/>
  <call process = "tns:VerifySeats"/>
</action>
```

3.4.2 Correlating the Action Context

The `correlate` element is used to define a relation between an action within the interface definition and a correlation definition. The purpose of this relation is to exactly indicate which properties serve to correlate an incoming message with a particular conversation, or more formally to indicate a particular execution context in which the action should be performed.

The syntax of the `correlate` element is:

```
<correlate correlation = QName
      instantiation = (true|false):false />
```

The `correlation` attribute references a correlation using fully qualified name. That allows referencing correlations defined in namespaces different from the namespace in which the interface is defined. This attribute is mandatory.

The `instantiation` attribute is optional and may have either value `true` or `false`. If the attribute has value `true` properties forming the correlation identity will be used to identify the current execution context in all subsequent message exchanges; the properties forming the correlation identity will be instantiated in that context. If the attribute has value `false` (the default) the correlation properties are used to identify a previously established execution context in which the action should be performed.

An action may specify zero or more correlate elements. These elements may have

different values for the instantiation attributes. If an action specifies, for instance, two correlate elements with the instantiation attributes having values `false` and `true` respectively, then:

- the first correlate element (`instantiation = false`) specifies the correlation whose properties serve to identify the execution context in which the action should be performed
- and the second correlate element (`instantiation = true`) specifies the correlation whose properties will be used to identify the same execution context in subsequent message exchanges.

An operation references a correlation whenever the message conveyed by the operation contains all the properties that form the correlation identity (as defined by the selectors that apply to such message). This applies to any WSCI action.

In order to associate an action with the proper context in which it is performed, the correlate element is used with the instantiation attribute set to false (or absent); in this case, the properties forming the correlation identity, must have been instantiated in that context before the correlation could be used in that way.

The properties forming the correlation identity can be instantiated:

- Upon the receipt of a message; this is expressed in the interface definition using the correlate element with the instantiation attribute having value true;
- By means not described by the WSCI interface; WSCI would not describe, in this cases, how the instantiation happens and each service implementation could differ; WSCI only enforces the properties qualifying the correlation and their types.

A context can be matched by means of an already established correlation only for operations that receive a message first, such as the WSDL one-way or request-response operations; the properties conveyed in the first message must be matched with properties that identify the context. The correlation identity can be conveyed, though, in any operation and individually in each message of that operation (both received and sent messages). Correlation properties can be instantiated by any operation that allows receiving of messages, such as a WSDL one-way, request-response and solicit-response operations.

The `correlate` element is not allowed for actions that can only send messages (such as the WSDL notification operation). An action that performs an operation that begins with sending a message followed by receiving a message (such as the WSCI solicit-response operation) may only specify the correlate element with the instantiation attribute having value true; the correlation properties, in this case, must be conveyed by the incoming message.

An action performing an operation that begins with an input message and is not correlated is used as part of the process instantiation, and in cases where all instances of the process are allowed to receive the message and there is no need to correlate the message to a specific instance, for example a broadcast message, which should be received by all process instances.

Example. This example shows a part of the behavior of the Traveler whose interface definition is completely specified in Section 5. The selected part of the behavior includes three actions: `BookTickets`, `ReceiveTickets` and `ReceiveStatement`. The first action performs a solicit-response operation and the last two actions perform one-way operations. In order to correlate the incoming messages (the confirmation of issued tickets and the statement) with the particular execution context (i.e. the context in which action `bookTickets` has been previously executed) the correlate elements are used. All three actions reference the same correlation `bookingCorrelation`.

The first action specifies the `correlate` element with the instantiation attribute having value `true`. Therefore the incoming message must carry values of the correlation properties and the property values are used to identify the execution context in which action `bookTickets` is performed in the remaining message exchanges.

```
<interface name = "Traveler">

   <process name = "PlanAndBookTrip">
      …
      <action name = "bookTickets"
                role = "tns:Traveler"
                operation = "tns:TravelerToTA/bookTickets">
         <correlate correlation = "defs:bookingCorrelation"
                instantiation = "true"/>
       </action>

      <all>
         <action name = "ReceiveTickets"
                role = "tns:Traveler"
                operation = "tns:TravelerToAirline/ReceiveTickets">
            <correlate correlation = "defs:bookingCorrelation"/>
         </action>

            <action name = "ReceiveStatement"
                  role = "tns:Traveler"
                  operation = "tns:TravelerToTA/ReceiveStatement">
            <correlate correlation = "defs:bookingCorrelation"/>
           </action>
       </all>
   </process>
</interface>
```

3.4.3 Calling a process from within an action

An action that handles a request-response message exchange can perform an arbitrary set of activities before it completes by calling another process. The call element is used to indicate the activities that would occur while the action is being performed by the service.

```
<call
   process = NCName>
   Content: (documentation?)
</call>
```

An action can perform an arbitrary set of activities only if the operational semantics indicate that such activities must occur between the beginning and end of the action. The call element is allowed when performing the WSDL *request-response* operation, but is forbidden for all other WSDL operations.

Since an action is an atomic activity, the process is invoked and completed before the action completes. If a fault occurs while performing that process, the action will also fault.

3.4.4 Extensibility of the action element

The `action` element is an extensible element. The `action` element can refer to operations defined in a specification other than WSDL using extension attributes. Extension attributes are defined in any namespace other than WSCI. The `operation` attribute and extension attributes are mutually exclusive.

The `action` element can specify additional semantics pertaining to the operation being performed. Additional semantics are expressed using extension elements that are defined in any namespace other than WSCI.

A WSCI processor may optionally process additional semantics in one or more namespaces. The manner in which such semantics are processed is outside the scope of this specification.

This specification does include one such extension that expresses the manner in which an action can identify a peer service against which an operation is performed. This extension consists of the locate elements which are defined later on in this specification. Both elements are defined in the WSCI/locate namespace (see Section 6.1)

3.5 Choreography Description

Atomic activities are the elementary units of the interface definition. An atomic activity is one that cannot be further decomposed and is performed in an all-or-nothing manner. The most common atomic activity is `action`. This activity will perform a single operation, e.g. receiving a message (*one-way* operation) or sending a message and waiting for a reply (*solicit-response* operation).

Complex activities are containers for sets of activities and define rules for coordinating the order in which activities are performed from within the activity set. Unlike atomic activities, complex activities can be decomposed into atomic and complex activities.

3.5.1 Activity

Interfaces define Web Services as performing activities of varying complexity. The definition of services as performing activities enhances the definition of services as performing operations by adding the proper context in which an operation is said to occur, and allowing operations to be associated over space and time. While operations are stateless in nature, activities are stateful.

As a result the interface can define a complex behavior that can include complex interactions involving multiple operations and services, transactional behavior and exception handling, and the dynamic establishment of links across services.

Similarly, an activity may be as simple as a one-way operation performed against one other service, or as complex as multiple message exchange with a set of services.

When two or more activities perform in the same context they often use the context as a container for shared properties. The activities can establish a relation to each other by appearing to read and write properties in that context. For example, an activity that sends a message can depend on a property established from a message received by a previous activity.

Contexts are also used to distinguish between multiple instances of the same activity. An activity not only establishes its dependency on a previous activity occurring in the same context, but on a specific instance of such activity. This form of continuity allows us to define complex stateful interactions involving the exchange of multiple messages with the same or different peer services.

A context must be established before the activity is performed. For that reason the context is not defined by the activity itself but ahead of the activity definition.

3.5.2 Activity Set

Since contexts are often used to establish a relation between two or more activities, we allow a context definition to be used by a set of activities. An activity set defines a set of activities together with the context in which they execute. The syntax for an *activity set* is:

```
Content: (context?,{any activity}+)
```

An activity set can include any type of activity element. This includes all activities

defined in this specification. In addition, activities defined in other specification can be used as long as they are defined in a namespace that is other than WSCI and use the `wsci:activity` abstract element as their substitution group.

Note that an *activity set* is not an element or a complex type, but rather a model group. It is used as part of the content of various elements, always in the form depicted above.

The `context` element is optional. If absent, we treat it as if the context definition as empty.

All activity elements are based on a common type definition with the syntax:

```
<{activity type}
   name = NCName>
   Content: (documentation?)
</{activity type}>
```

The optional `name` attribute provides the non-qualified activity name to distinguish the activity from any other activity defined in the same context. If used, the activity name must be different from any other entity defined in the same context, including other activities, properties, nested processes, etc (see Section 3.5.4 for more details about contexts).

An activity is always performed within some context. The definition of a context makes the distinctive difference between stateless operations and stateful actions. The context in which an activity is performed is referred to as the activity's current context.

3.5.3 Complex Activity

A *complex activity* is any activity that contains one or more activity sets. The complex activity defines the rules for determining which activity set will be performed, determining the number of times the activity set will be performed, and determining the order in which the activities will be performed from the set:

- The complex activity `all` operates on a single activity set and performs the activity set exactly once; the activities can be performed in non-sequential order.
- The complex activity `sequence` operates on a single activity set and performs the activity set exactly once in sequential order.
- The complex activity `choice` selects one activity set based on a triggered event, and performs that activity set exactly once in sequential order.
- The complex activity `foreach` operates on a single activity set and performs

the activity set zero or more times, performing the activities in sequential order in each iteration.

- The complex activity `switch` selects one activity set based on a condition, and performs that activity set exactly once in sequential order.

3.5.4 Context

Just like activities, contexts are composed recursively. An activity is performed in the current context, as defined by the activity set in which the activity appears. When that activity set is contained in another activity, that parent activity provides a parent context. The current context encompasses the parent context and all its parent contexts recursively.

In fact, the current context is a combination of the parent context and any local declaration made in that context. A local declaration is a declaration made in the current context that hides any entity with the same name in the parent context. A local declaration can be used to confine an entity to the current context, or to hide changes that could occur in the current context from affecting any parent or sibling context.

The syntax for a context definition is:

```
<context>
   Content: ((process | property)*,exception?,
            transaction?)
</context>
```

- **Property:** Declaring a property as local to a context assures that any changes to the value of that property (as a result of activities occurring in that context) do not affect the value of that property in any parent context.

- **Exception:** Declaring an exception behavior as local to a context assures that that definition will be honored in case of an exception occurring in that context, without changing the definitions prescribed for a parent context.

- **Process:** Declaring a process as local to a context (also known as nested process) assures that this particular process definition is not available in the parent or sibling context, and further allows the nested process to establish a relation with any other activity or process occurring in the same context.

- **Transaction:** Declaring a transaction as local to a context assures that all activities occurring in that context and only these activities are performed as part of that transaction.

Properties and processes declared within a given context must use unique names. It is an error for two property declarations, process declarations or a property and

process declaration in any given context to have the same fully qualified name.

A context is established before performing the first activity in the activity set and is discarded after performing the last activity in that activity set. A context is not established until it has not been determined that at least one of the activities in the activity set are to be performed. The declarations of a context are available to the set of activities while the context is active, i.e. between establishing and discarding the context.

The syntax for a local property declaration is as follows:

```
<property
   name = QName
   select = XPath>
   Content: (documentation?, value?)
</property>

<value>
   Content: {mixed}
</value>
```

The `name` attribute provides the property name. The property name is a fully qualified name since it is permissible to use properties defined in a namespace other than the namespace used for the interface definition, in particular properties that are shared across many interface definitions. If no other element or attribute is used, the property value is obtained from the property with the same name in the parent context.

The `select` attribute is an XPath expression. If used, the expression is evaluated to arrive at the value of the property. This attribute is used to establish a dynamic value that can differ across instances of the same context.

The `value` element is a contained or mixed contents. If used, the specified value is used as the value of the property. This attribute is used to establish a static value that is identical across instances of the same context. The value can include elements in any namespace and its content need not validate.

The `select` attribute and `value` element are mutually exclusive; it is an error to use both at the same time. In either case, changes to the named property that occur within the context in which it is defined do not affect any property with the same name in any parent or sibling context.

3.6 Choreography Elements

3.6.1 All

The `all` activity is a complex activity. If performs all the activities within the activity

set in non-sequential order, possibly in parallel.

```
<all
   name = NCName>
   Content: (documentation?,context?,{any activity}*)
</all>
```

3.6.2 Choice

The `choice` activity is a complex activity that selects one activity set from a collection of two or more activity sets based on the first event triggered.

```
<choice
   name = NCName>
   Content: (documentation?,
           (onMessage|onTimeout|onFault){2,n})
</choice>

<onMessage>
   Content: (documentation?,action,
             context?,{any activity}+)
</onMessage>

<onTimeout
   property = QName
   type = (duration|dateTime) : duration
   reference = QName>
   Content: (documentation?,context?,{any activity}+)
</onTimeout>

<onFault
   code = QName>
   Content: (documentation?,context?,{any activity}+)
</onFault>
```

The `choice` activity must specify two or more mutually exclusive events.

Events that may be defined with a choice are receipt of a message, the raising of a fault, and the elapsing of a time-out.

If two or more events overlap with each other the interface is considered ambiguous.

3.6.2.1 Example

In this example (taken from Section 5.4.3.2), the Airline Reservation System describes how it waits for one of two operations to be performed. Either booking is requested (by means of the `BookSeats` operation), or the reservation is cancelled (by means of the `CancelReservation` operation). In the first case, the service will proceed by issuing the tickets and sending them to the Traveler. In the second

case, the service will proceed by compensating for the reservation transaction.

```
<choice>
      <onMessage>
        <action name = "PerformBooking"
                role ="tns:Airline"
                operation = "tns:AirlineToTA/BookSeats">
          <correlate correlation="defs:reservationCorrelation"/>
        </action>

         <action name="SendTickets"
                role ="tns:Airline"
                operation="tns:AirlineToTraveler/SendTickets"/>
      </onMessage>

      <onMessage>
        <action name = "ReceiveCancellationRequest">
                role ="tns:Airline"
                operation="tns:AirlineToTA/CancelReservation">
          <correlate correlation="defs:reservationCorrelation"/>
        </action>
        <compensate name = "CompensateReservation"
                    transaction = "tns:seatReservation"/>
      </onMessage>
</choice>
```

3.6.2.2 Message Event Handler

The `onMessage` event handler responds to an incoming message. The initial action defines the event that triggers this event handler. If this atomic activity is performed successfully, the activity set is performed. If this atomic activity faults, the `choice` activity will fault with the same code.

The initial action is allowed to perform the WSDL *one-way* or *request-response* operation, but is forbidden to perform all other WSDL operations.

3.6.2.3 Timeout Event Handler

The `onTimeout` event handler responds to a time-out event. The `property` attribute is the name of a property, which holds the time-out value. The type attribute specifies the type of the time-out value as either `dateTime` or `duration`:

- `dateTime` The named property specifies the date/time instant at which the time-out event occurs. The XML Schema datatype dateTime can be used to specify a date/time instant value.

- `duration` The named property specifies the time duration relative to a reference time. The XML Schema datatype duration can be used to specify a duration value.

The default reference time for a time-out of type duration is the time instant at which the `choice`activity was started. The time-out date/time instant is calculated by adding the duration to the reference time.

The `reference` attribute is used to specify a different reference time. This attribute can be used only if the time-out type is duration. The `reference`attribute is the name of a property that holds a reference date/time instant. The XML Schema datatype dateTime can be used to specify a date/time instant value.

The `reference` attribute can refer to the start or end time of an activity, transaction or process using the suffix @start or @end, respectively. The referenced entity must be performed in the same or parent context in which it is referenced.

The syntax for the `reference` attribute is:

```
reference =: property | entityStart | entityEnd
property =: QName
entityStart =: QName'@start'
entityEnd =: QName'@end'
```

3.6.2.4 Fault Event Handler

The `onFault` event handler responds to a fault. The code attribute specifies the fault code. If used, the event handler only responds to a fault with this code. If absent, the event handler responds to all faults for which no other event handler is specified.

The `choice` activity defines the behavior when a fault occurs in the service, while waiting for another event to occur, e.g. receipt of a message. Faults that occur when performing an action, e.g. a WSDL solicit-response operation, and faults that occur when performing activities (see the fault activity) are caught by exception event handlers.

3.6.3 Behavior

The first event to occur will trigger the corresponding event handler and perform the activity set specified by that event handler. All other event handlers will be ignored.

The event handlers specified by `choice` may overlap with event handlers specified by `exception` and will take precedence in case of an overlapping event.

3.6.4 Foreach

The `foreach` activity is a complex activity that performs all the activities within the activity set repeatedly, once for each item in a pre-selected list.

```
<foreach
   name = NCName
   select = expression>
   Content: (documentation?,context?,{any activity}+)
</foreach>
```

The `select` attribute specifies an expression that evaluates to a list of zero or more items. This attribute is an XPath expression. The item list is selected before the activity set is performed. The activity set is repeated once for each item in the selected list in the same order in which the list items are constructed by the XPath expression, or zero times if the list is empty.

For each iteration the value of the current item is held in the property `wsci:current` which is local to the context in which the activity set is performed.

3.6.5 Sequence

The `sequence` activity is a complex activity that performs all the activities within the activity set in sequential order.

```
<sequence
   name = NCName>
   Content: (documentation?,context?,{any activity}+)
</sequence>
```

3.6.5.1 Example

In this example (taken from Section 5.4.1.2), the Travel Agent service describes how some operations must be performed in sequential order; first, the Travel Agent indicates an intention to cancel the reservation by sending a cancellation request to the airline. It, then, waits for receipt of a cancellation confirmation from the airline. Last, it notifies the traveler that the reservation has been cancelled.

```
<sequence>
  <action name = "CancelReservation"
          role = "tns:TravelAgent"
          operation = "tns:TAtoAirline/RequestCancellation"/>

   <action name = "ReceiveCancellationNotification"
          role = "tns:TravelAgent"
          operation = "tns:TAtoAirline/AcceptCancellation">
    <correlate correlation = "defs:reservationCorrelation"/>
  </action>

  <action name = "NotifyOfCancellation"
          role = "tns:TravelAgent"
          operation="tns:TAtoTraveler/NotifyOfCancellation"/>
</sequence>
```

3.6.6 Switch

The `switch` activity is a complex activity that selects one activity set from a collection of one or more activity sets based on the evaluation of one or more conditions.

```
<switch
   name = NCName>
   Content: (documentation?,case+,default?)
</switch>

<case>
   Content: (documentation?,condition,
          context?,{any activity}+)
</case>

<default>
   Content: (documentation?,context?,{any activity}+)
</default>

<condition
   {extension attribute}>
   Content: {expression}
</condition>
```

The `case` element selects an activity set based on the truth-value of a condition. All cases are mutually exclusive.

The order in which `case` elements are specified is important. The conditions are evaluated in that order, the first condition to be met will cause the activity set to be performed ignoring all other cases.

The `default` element selects an activity set in the event that no other condition has been met. If absent and no condition is met the switch activity completes without executing any activity.

The `condition` element specifies a condition that must evaluate to a Boolean value, as defined by the XPath specification. The condition is evaluated in the context in which the switch *activity* is performed. Conditions are most commonly expressed using the XPath language.

The `condition` element is an extensible element. If an extension attribute is used the type of expression and the manner in which it is processed depends on the extension attributes. Extension attributes are defined in any namespace other than WSCI.

3.6.6.1 Example

In this example (taken from Section 5.4.2.2), the condition is abstract and does not indicate the internal logic employed by the Traveler, as this may differ across implementations. The Traveler selects between canceling an itinerary and

reserving tickets.

```
<switch>
   <case>
     <condition>tns:cancelItinerary</condition>
     <action name = "CancelItinerary"
          role = "tns:traveler"
          operation = "tns:TravelerToTA/CancelItinerary"/>
  </case>
  <default>
    <action name = "ReserveTickets"
          role = "tns:traveler"
          operation = "tns:TravelerToTA/ReserveTickets"/>
  </default>
</switch>
```

3.6.7 Until

The `until` activity is a complex activity that performs all the activities within the activity set repeatedly, one or more times, based on the truth value of a condition.

```
<until
name = NCName>
Content: (documentation?,condition,
          context?,{any activity}+)
</until>
```

The activity set is performed at least once. The condition is evaluated after each completion of the activity set. If false the activity set is performed again, otherwise the until activity completes.

Refer to the `switch` activity for the definition of the `condition` element.

3.6.8 While

The `while` activity is a complex activity that performs all the activities within the activity set repeatedly, zero or more times, based on the truth-value of a condition.

```
<while
name = NCName>
Content: (documentation?,condition,
          context?,{any activity}+)
</while>
```

The condition is evaluated repeatedly before the activity set is performed. If true the activity set is performed, otherwise the `while` activity completes. Refer to the *switch* activity for the definition of the *condition* element.

3.7 Other activities

3.7.1 Delay

The `delay` activity is an atomic activity expressing a time interval.

```
<delay
   name = NCName
   property = QName
   type = (duration|dateTime) : duration
   reference = refType>
   Content: (documentation?)
</delay>
```

This activity completes after the specified delay. It may complete prior to the specified delay if an exception occurs in the context in which the delay activity is defined, e.g. a fault or a message received by an exception event handler.

The `property`, `type` and `reference` attributes specify the delay as either time duration or a date/time instant, and are used as with the `onTimeout` event handler (see there) with the following difference: the default reference time for a delay of type duration is the time instant at which the delay activity started.

3.7.2 Empty

The `empty` activity is an atomic activity that does nothing.

```
<empty
   name = NCName>
   Content: (documentation?)
</empty>
```

The `empty` activity can be used in places where an activity must appear, but the behavior of the interface does not reflect any observable activity (for instance when performing an internal operation).

3.7.3 Fault

The `fault` activity is an atomic activity that triggers a fault in the current context.

```
<fault
   name = NCName
   code = QName>
   Content: (documentation?)
</fault>
```

The `code` attribute specifies the fault code. The fault will trigger the matching `onFault` exception event handler in the same context in which the fault occurs.

If the fault is not handled by any exception event handler defined in the current

context, or in any parent context, the process will complete with that fault code. If the process is called by an action, e.g. when performing the WSDL request-response operation, the action will complete with that fault code, returning a fault message to the sender.

3.8 Composition and Re-use

WSCI also supports the definition of interfaces that express complex processes, and can, in essence, handle multiple 'threads' of execution simultaneously. Multiple parallel 'threads' with differing choreographies are specified with the 'spawn/join' construct.

Those with the same choreography are specified implicitly with the 'correlation' construct. Correlation specifies the set of properties that collectively allow the service provider to connect any incoming message to the right 'instance' of the choreography.

Furthermore, each action/activity can be described as an inline definition in a parent activity or process, or it can be reused by defining it as a process and referring it with the call activity.

3.8.1 Process

A `process` is a special type of activity that establishes its own context of execution. As such it can serve as a top-level definition that is directly accessible for instantiation; a process does not need a pre-established context to execute in, but it instantiates its own context as it is instantiated.

A process defines one activity set that is performed exactly once and all activities in that activity set are performed in sequential order. A process can be instantiated from another process by spawning or calling it. A process can also be instantiated from another service by sending it a message, when the process is defined as being instantiated upon receipt of a message.

```
<process
   name = NCName
   instantiation = message | other : message>
   Content: (documentation?, context?, {any activity}+)
</process>
```

The `name` attribute provides the non-qualified process name to distinguish the process definition from any other process defined in the same interface or context.

The `instantiation` attribute indicates whether the process is instantiated in response to a message or by other means.

If the value of the `instantiation` attribute is `message`, the process is instantiated

upon receipt of a message. The process definition must begin with an activity that responds to an incoming message, e.g. an action that references a WSDL *one-way* or *request-response* operation.

The `process` definition is allowed to begin with an `all` activity that includes only such actions. In this case the process will be instantiated upon receipt of all messages at once. It is intended that all actions will complete upon instantiation of the process.

The `process` definition is allowed to begin with a `choice` activity that consists only of `onMessage` event handles that begin with such an action. In this case the process will be instantiated upon receipt of any one message. It is intended that exactly one action will complete upon instantiation of the process.

If the value of the `instantiation` attribute is `other`, the process must be instantiated by some other means.

It is possible to instantiate such a process using the `spawn` and `call` activities, but only from other processes defined in the same interface. Other means of instantiation are possible, but not covered by the WSCI specification.

When a process is instantiated it establishes the context and performs all the activities in the activity set in sequential order. In that respect a process definition is similar to the `sequence` activity, but it does not have to be contained within an activity set.

If the `process` definition appears within the `interface` element, we consider it a top-level process definition. A top-level process is performed in a context that is independent of any other context.

If the `process` definition appears within the `context` element, we consider it a nested process definition. A nested process is performed in the context in which it is defined. Unlike a top-level process, the nested process uses the context in which it is defined as its parent context, and may share the same properties as its parent process. A nested process may require local property declarations to isolate itself from any other activity performed in that context.

When `instantiation` type is `other`, a nested process may be instantiated from any activity occurring in the same context in which it is defined, a child context, or another nested process in the same context. When `instantiation` type is `message`, a nested process may be instantiated only as part of an instance of that context.

A nested process may complete after all activities defined in the same context have completed. For that reason, event handlers that are defined in that context do not react to any exceptions that occur in the nested context. The parent context may use the join activity or a transaction to establish such a relationship.

3.8.2 Call

The `call` activity is an atomic activity that instantiates a process and waits for it to complete.

```
<call
   name = NCName
   process = NCName>
   Content: (documentation?)
</call>
```

The named process definition must be visible in the current context in which the call activity is performed.

The called process is instantiated in the same context in which it is defined. This context may be different from the context in which the call activity is performed.

The call activity waits until the instantiated process completes. If the process faults, the call activity will fault with the same code. The call activity does not directly affect any call, spawn or join activity that occurs in the same or different context.

3.8.3 Spawn

The `spawn` activity is an atomic activity that instantiates a process.

```
<spawn
   name = NCName
   process = NCName>
   Content: (documentation?)
</spawn>
```

The `process` attribute names the spawned process. The named process definition must be visible in the current context in which the spawn activity is performed. The spawned process is instantiated in the same context in which it is defined. This context may be different from the context in which the spawn activity is performed. The spawn activity creates a new process instance and completes. It does not wait for the spawned process to perform any activity.

3.8.4 Join

The `join` activity is an atomic activity that waits for instances of the spawned process to complete.

```
<join
   name = NCName
   process = NCName>
   Content: (documentation?)
```

```
</join>
```

The `process` attribute names the process to join. This activity waits for the completion of all instances of the named process that were spawned in the current context and have not completed yet. This includes instances that were instantiated by a message, by the spawn activity, or by some other means. Instances that were instantiated by the call activity cannot be joined. If no instance exists, or all instances have already completed, the activity completes.

3.9 Exception Handling

Activities define the expected behavior of a service. Unexpected behavior is captured by defining exception handling. Exception handling is associated with a particular activity set by defining it as part of the context in which the activity set is performed. Since contexts are composed hierarchically, exception handling behavior defined in one context can be used to mask exception handling behavior defined in a parent context, and in reverse, exception handling behavior defined in a parent context can act as the default behavior for all child contexts.

3.9.1 Exception

Exception handling is associated with a particular activity set by defining it as part of the context in which the activity set is performed. Since contexts are composed hierarchically, exception handling behavior defined in one context can be used to mask exception handling behavior defined in a parent context, and in reverse, exception handling behavior defined in a parent context can act as the default behavior for all child contexts.

Exception handling behavior is defined using the `exception` element:

```
<exception>
   Content: ((onMessage | onTimeout | onFault){+})
</exception>
```

The `exception` handler must specify one or more event handlers. An event handler defines the event and the activity set to perform, should that event occur.

Events are mutually exclusive; if two or more events overlap with each other, or if the `onMessage` event overlaps with an action in the same or child context, the interface is considered ambiguous.

The `onMessage`, `onFault` and `onTimeout` event handlers are similar to those used for the `choice` activity (see there), with the following differences:

- `onMessage`: If this atomic activity faults, the exception handler will fault with the same code.

- `onTimeout`: The default reference time for a time-out of type duration is the time instant at which the context was established. The time-out date/time instant is calculated by adding the duration to the reference time.

- `OnFault`: The exception event handler will response to a fault occurring while performing an action in that context, e.g. a WSDL solicit-response operation. The *fault* activity can be used to force the presence of a fault not triggered by the execution of an action. If a fault is not caught by an exception event handler within the context in which it occurs, the fault will be propagated to the parent context. If the fault is not caught in any parent context, the process will complete with that fault code. If the process is called by an action, e.g. when performing the WSDL request-response operation, the fault will be returned to the sender.

The first event to occur will trigger the corresponding event handler and perform the activity set specified by that event handler. All other event handlers defined in the same exception handler will be ignored. However, event handlers defined in a parent context are still in effect and may respond to exceptional events occurring while performing any event handler in the child context.

The activities performed by the activity set are completed before the activities defined in the event handler are performed.

The service must wait until it can either complete in-progress atomic activities, or cancel such activities, behaving as if they never occurred. It is an error to behave as if an atomic activity has partially completed.

The service is able to partially complete an in-progress complex activity by completing all in-progress activities in its activity set and not performing any further activities in the activity set. The activity set context is discarded before the complex activity completes. The context in which the exception handler appears is discarded only after the activity set specified by the event handler has completed.

3.10 Transactional behavior

3.10.1 Transactions

WSCI Transactions are used to model the behavior of a Web service asserting that a certain number of activities should be treated as a single unit of work; a Web service uses a WSCI transaction to communicate to other services its ability to either completely execute those activity or to restore the consistent state prior to the execution.

A Web service may, also, use a WSCI transaction to communicate to other services its ability to perform the transactional block as part of a distributed transaction protocol.

3.10.1.1 Atomic Transactions

A WSCI Atomic transaction is used to model an ACID transaction. Atomic transactions assure that all activities performed as part of the transaction will behave as a single unit of work. If the transaction cannot complete successfully, it will rollback to the state before the beginning of the transaction.

Individual activities can be atomic, but there is no guarantee that all activities will complete successfully or rollback. The atomic transaction gives an all-or-nothing guarantee to any activity set that is performed as part of the transaction.

Atomic transactions require resource locking. Resource locking can be disruptive if locks are maintained for a long period of time, as such, atomic transactions are only recommended for short-lived transactions.

3.10.1.2 Open Transactions

Open (Nested) transactions relax the isolation requirement of atomic transactions and allow arbitrary levels of nesting. With open transactions, resources are acquired for short periods of time and then released, typically by using a combination of nested open and atomic transactions. As such, open transactions can be used for long-lived transactions that cannot complete in a short time span.

Open transactions allow activities to progress from one consistent state to another, making each change permanent and durable immediately upon completion of the activity. As a result, open transactions are more resilient to system failures and are useful in supporting long-lasting transactions.

In the event of a system failure, an atomic transaction will rollback and must be retried. This approach is practical for transactions that can be performed in a short time period. In contrast, an open transaction will continue to perform its activities past a point of failure, and can be used for transactions that span an arbitrary time period.

Open transactions require additional work in order to perform backward recovery, since the effects of the transaction cannot be reverted automatically. Often, this is accomplished by compensating for the nested transactions in order to rollback the parent transaction.

Open transactions attempt to provide an all-or-nothing guarantee, but may not achieve full atomicity. Unlike atomic transactions, open transactions can be used in cases where the effects of a transaction cannot be reverted completely as part of a rollback or compensation.

3.10.1.3 Compensation

An open transaction made of multiple activities and transactions, rolls back by reverting their effects. Whilst rollback occurs while the transaction aborts and cancels the effects of the transaction that has not yet completed, compensation occurs after the transaction completes and reverts the effects of the completed transaction.

Compensation defines the logic for reverting the effects of a completed activity or transaction.

A parent transaction fulfills its responsibility to compensate by providing its own set of activities, and by invoking the compensating activity set of a nested transaction using the `compensate` activity.

In WSCI, the transaction definition can specify which activities will occur when it is compensated; these activities will be performed when invoked by the `compensate` activity.

The compensating activities (after completion) are defined separately from the rollback (before completion); both are defined together with the transaction as part of the same WSCI context.

A transaction instance can only be compensated once.

3.10.1.4 Behavior

Any activity set may be assigned transactional behavior by defining its context using the transaction element. All activities executed in that context, including atomic and complex activities and nested processes instantiated in that context, are performed as part of that transaction. The activity set cannot complete until the transaction has completed or aborted.

If a context defines its own transaction and has a transactional parent context, that transaction is a nested transaction. The parent-child relationship of contexts transfers to a nested relationship between the transactions. Similarly, a transaction is nested if it is defined as part of a nested process that itself is defined in a transactional context.

Both atomic and open transactions can be nested in an open transaction. It is an error to nest any transaction within an atomic transaction, since protocols that support atomic transactions do not support nesting of transactions.

A transaction can initiate recovery by catching an exception within an exception event handler that is defined in the transaction context. The transaction event handler indicates which activities will occur when the transaction attempts to rollback. The semantic of rollback is only applied to an event handler if used in a transactional context.

The `compensate` activity is used to compensate for a transaction. This activity can only be performed in a parent context of the completed transaction; the compensation behavior is executed in the context where the completed transaction executed.

The compensation activity set may reference local property declarations made in the compensation context and local property declarations made in the transaction context. All other properties are referenced in the parent context of the transaction.

A nested transaction can complete or abort independently of the parent transaction. If the nested transaction has completed and the parent transaction aborts, it is the responsibility of the parent transaction to compensate for the nested transaction.

A parent transaction is not allowed to complete until all nested transactions either complete or abort. A nested process that is defined in a transactional context (or any of its child contexts) is instantiated as part of that transaction. A transaction is not allowed to complete until all activities performed in that transaction have completed, including any nested processes defined in the transaction context. A nested process is defined in a transactional context to indicate that a fault of the nested process causes the transaction to abort.

3.10.1.5 Syntax

The syntax for a `transaction` definition is:

```
<transaction
   name = NCName
   type = atomic | open : atomic
   retries = QName>
   Content: (compensation?)
</transaction>
```

The `name` attribute provides the transaction name. The type attribute specifies the transaction type. The default, when the `type` attribute is missing, is to describe an atomic transaction.

The `retries` attribute is a property that specifies the number of times the transaction can be repeated until it completes. If absent, zero is assumed, indicating that the transaction will be performed exactly once.

The `compensation` element is optional. If missing, then no activities will occur when the transaction is compensated.

The syntax for the transaction `compensation` definition is:

```
<compensation>
```

```
    Content: (documentation?, context?, {any activity}+)
</compensation>
```

3.10.1.6 Example

In this example (taken from Section 5.4.3), the atomic transaction performs seat reservation. The transaction begins upon receipt of the first reservation request and terminates with receipt of the last reservation request. The last reservation request is identified by having the `defs:notLastSeat` property set to `false`. In many cases a single reservation is made, but the transaction allows multiple reservations to be made as part of a single unit of work. The transaction completes only if all reservations were successfully completed.

```
<sequence>
  <context>
    <transaction name = "seatReservation"
                 type = "atomic">
      <compensation>
        <action name = "NotifyOfCancellation"
                role = "Airline"
                operation = "tns:AirlineToTA/NotifyOfCancellation"/>
      </compensation>
    </transaction>
  </context>
  <action name="ReserveSeat"
              role ="Airline"
              operation="tns:AirlineToTA/ReserveSeat"/>
      <while name="ReserveSeats">
        <condition>defs:notLastSeat</condition>
        <action name="ReserveSeat"
                role ="Airline"
                operation="tns:AirlineToTA/ReserveSeat">
          <correlate correlation = "defs:reservationCorrelation" />
    </action>
  </while>
</sequence>
```

The compensating behavior for the transaction is specified as part of the transaction and could be invoked as part of exception handling defined in a following context:

```
<exception>
  <onTimeout property = "tns:expiryTime"
        type = "duration"
        reference="tns:ReserveSeats@end">
     <compensate transaction = "tns:seatReservation"/>
  </onTimeout>
</exception>
```

3.10.2 Compensate

The `compensate` activity is an atomic activity which performs compensation for any instances of the named transaction.

```
<compensate
   name = NCName
   transaction = NCName>
   Content: (documentation?)
</compensate>
```

The `transaction` attribute specifies the transaction to compensate. The name must match a previously completed transaction that was performed in the current context.

The activity performs compensation for any instances of the named transaction that occurred in the current context, have completed and have not been compensated for. A transaction instance is not compensated if the transaction has not completed successfully, or if the transaction instance has already been compensated for.

3.11 Interface

The `interface` element describes the WSCI view of a Web Service participating in a choreographed, long-lasting and stateful message exchange with other services. The syntax of the interface element is as follows:

```
<interface
   name = NCName>
   Content: (documentation?, process+)
</interface>
```

The `name` attribute of the interface element is required and allows the interface to be referenced from the global model. The interface element must contain one or more process elements that jointly describe the details of the message exchange.

The interface is a top-level WSCI definition. Therefore, the interface element appears as child of the `wsdl:definitions` element.

3.11.1.1 Example

This example shows part of the interface of the Travel Agent in the Ticket Reservation Process:

```
<wsdl:definitions>
  <wsci:interface name = "TravelAgent" />
    <documentation> … </documentation>
    <process name = "PlanAndBookTrip" instantiation = "message">
      <!--process details -->
    </process>
```

```
  </wsci:interface>
  <!--other definitions, e.g. correlations, selectors -->
</wsdl:definitions>
```

The detailed behavior of the Travel Agent in the Ticket Reservation Process is described inside the process `PlanAndBookTrip`. It is not shown in the XML fragment above.

---

4. Global model

The WSCI Global Model allows describing a message exchange oriented view of the overall process in terms of interfaces representing participants in that process, and links between operations of communicating participants. These links indicate direct message flow between the linked operations. The WSCI Global Model is a useful tool for visual modeling, analysis, validation and simulation of the overall process. The syntax for the WSCI Global Model is as follows:

```
<model name = NCName>
   Content: (documentation?,interface{2,n},connect+)
</model>

<interface ref = QName/>

<connect operations = twoOpName>
   {extension attribute}
   Content: {extension element}?
</connect>

twoOpName = list of opName {2}
opName = portTypeName'/'operationName
portTypeName = QName
operationName = NCName
```

The `model` element serves as a container for the WSCI Global Model. The `model` element is a top-level definition in WSCI. The `name` attribute is required. The `model` element must contain at least two references to interfaces of participants, and one or more links between operations of communicating participants.

The `interface` element references a WSCI interface. The value of the required ref attribute is a `QName` and must match the name of the referenced interface.

The `connect` element describes a link between operations of communicating participants. Usually, those operations will be mirror images of each other. The operations attribute serves the purpose to connect two WSDL operations. The definition of the operations attribute is specifically designed to work with WSDL since the `twoOpName` simple type is aligned with the naming conventions for WSDL port types and operations. If two operations defined in any other service

description language are to be connected, some extension of the connect element must be used.

Example.

The following XML fragment shows part of the global model for the Ticket Reservation Process. The full example is presented in Section 5. of this specification.

```
<wsdl:definitions   name = "Traveler"
   targetNamespace = "http://example.com/consumer/models"
        xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
        xmlns = "http://www.w3.org/2002/07/wsci10"
        xmlns:tra = "http://example.com/consumer/traveler"
        xmlns:air = "http://example.com/consumer/airline"
        xmlns:ta = "http://example.com/consumer/travelagent">

   <wsdl:import   namespace = "http://example.com/consumer/traveler"
          location = "http://example.com/traveler.wsci" />
   <wsdl:import   namespace = "http://example.com/consumer/airline"
          location = "http://example.com/airline.wsci" />
   <wsdl:import   namespace = "http://example.com/consumer/travelagent"
          location = "http://example.com/travelagent.wsci" />

   <model name = "AirlineTicketing">

     <interface ref = "air:Airline" />
     <interface ref = "tra:Traveler" />
     <interface ref = "ta:TravelAgent" />

     <!-- Traveler / TravelAgent -->

     <connect operations =    "tra:TravelerToTA/PlaceItinerary
                               ta:TAtoTraveler/ReceiveTrip" />

     <!-- other connect elements -->

   </model>
</wsdl:definitions>
```

The Global Model presented above contains references to the interfaces of the three participants in the Ticket Reservation Process. The interfaces are referenced by their qualified names. In addition, the Global Model describes how operations that directly map a message flow are actually connected. In the above example, the Global Model describes how the `PlaceItinerary` operation defined within the `TravelerToTA` port type and the `ReceiveTrip` operation defined within the `TAtoTraveler` port type are connected and, together, describe how the trip order is sent by the Traveler and received by the Travel Agent. For a full description of the Ticket Reservation Scenario please refer to Section 5.

5. Example

5.1 Overview

This section provides a comprehensive example that illustrates how WSCI can be used to model a scenario that better reflects the real life business process of reserving and booking airline tickets.

This example expands on the simple one introduced in Section 1.7, so that it will highlight the use of each construct of the WSCI syntax (as described in Section 3).

The scenario now includes three participants: a Traveler, a Travel Agent, and an Airline Reservation System. This section will guide through the definition of the WSCI interfaces of each participant as they collaborate in the overall process called `PlanAndBookTrip`. The actual implementation of these interfaces is up to each party to decide; WSCI is completely agnostic on this. Perhaps the Traveler is using a browser-based plug-in web service, like a specialized applet or a mail agent. Perhaps the travel agent has it own tailored web service. Perhaps the airline has a web service that comes built into its overall reservation application.

In the following, picture, the dotted circle highlights the scope of this example, namely the definition of a WSCI interface for each participant, and the global model describing how those interfaces interact.

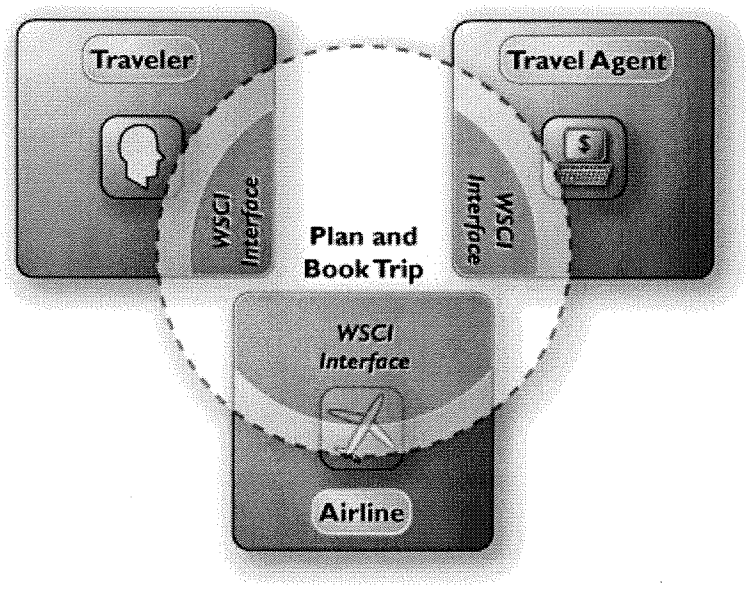


Figure 5-1

5.2 Use Case definition

The high level flow of the messages exchanged within the "*Plan And Book Trip*" process can be visualized as per the following picture:

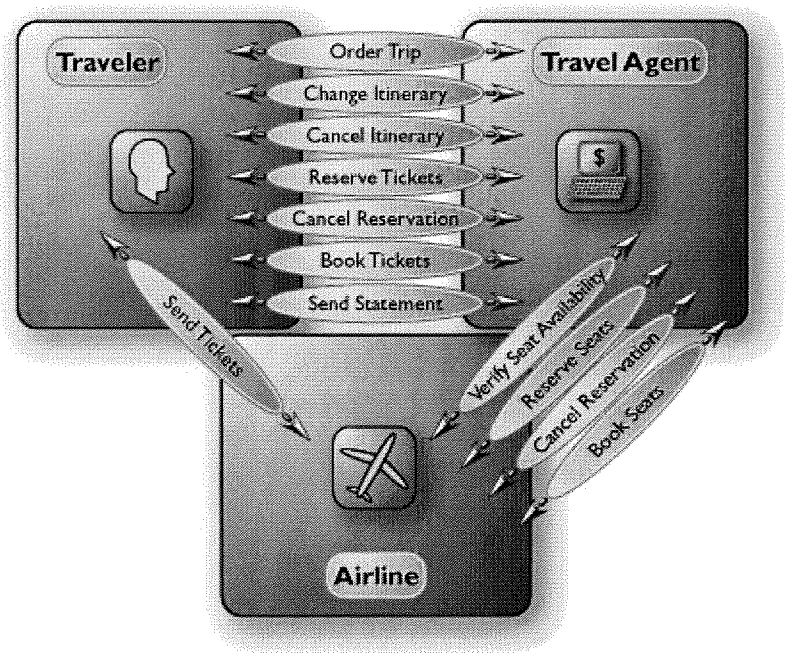


Figure 5-2

The following description of the scenario makes use of the step labels shown in the picture.

A Traveler is planning on taking a trip. She decides where to go (her final destination) and when to leave and return. These activities are performed independently on any specific contact the Traveler may have with the Travel Agent (for instance, the trip may be planned using a catalogue or any other means which do not involve connecting to the Travel Agent site).

Once she picks her preferred plan, she submits her choice to a Travel Agent by means of her local Web Service software (*Order Trip*).

The Travel Agent evaluates which is the best itinerary to reach the desired destination, based on the Traveler's criteria such as cheapest price, availability of destination, type of plane or frequent flyer miles. The Travel Agent finds that the best travel plan includes one or more discrete journeys, or legs which, for the sake of simplicity, will all be operated by the same Airline.

For each leg, the Travel Agent asks the Airline Reservation System to verify the availability of seats (*Verify Seats Availability*). This availability is constrained by the departing/arrival time information and by the Traveler's seat choice. For each leg, the Airline Reservation System provides information about the availability of a

particular seat. Note that the Travel Agent may have to restart the whole selection in case the Airline would not be able to verify the availability of a given seat.

Once the ideal travel plan has been validated, the Travel Agent builds a proposed itinerary for the Traveler to verify. This itinerary might actually not be satisfactory to the Traveler for various reasons. So, the Traveler has the choice of accepting or rejecting the proposed itinerary as well as she may decide not to take the trip at all.

- In case she rejects the proposed itinerary, she may submit the modifications (*Change Itinerary*) and wait for a new proposal from the Travel Agent.
- In case she decides not to take the trip, she informs the Travel Agent about her decision (*Cancel Itinerary*) and the process ends.
- In case she decides to accept the proposed itinerary (*Reserve Tickets*), she will provide the Travel Agent with her Credit Card information in order to properly book the itinerary. The Traveler will, also, provide her contact information for the Airline Reservation System since the Traveler wants to receive an e-Ticket directly from the Airline.

Next, the Travel Agent connects with the Airline Reservation System in order to finalize the seat reservation (*Reserve Seats*). The Airline Reservation System electronically reserves the seats associated to each leg; the validity period for such reservation is of one day meaning that if a final confirmation will not be received within one day, tickets will be unreserved and the Travel Agent will be notified.

Next, the Travel Agent informs the Traveler about the reservation of the seats.

The Traveler can now either finalize the reservation or cancel it. If she confirms the reservation (*Book Tickets*), the Travel Agent asks the Airline Reservation System to finally book the seats (*Book Seats*). The Airline Reservation System books the seats for the chosen itinerary and, then, issues an e-ticket to the Traveler (*Send Tickets*).

Finally, the Travel Agent charges the Traveler's Credit Card with the relevant amount and sends the notification of the charge (*Send Statement*) together with a detailed description of the itinerary to the Traveler.

5.3 Building from the simple example

The three participants will define their respective WSCI interfaces for this process (see Section 5.4). To properly interact, the interfaces should form 'mirror images' of each other, so that the WSCI action of one is 'matched' with a corresponding WSCI action of another. In this example, thus, the overall picture looks like the

following figure.

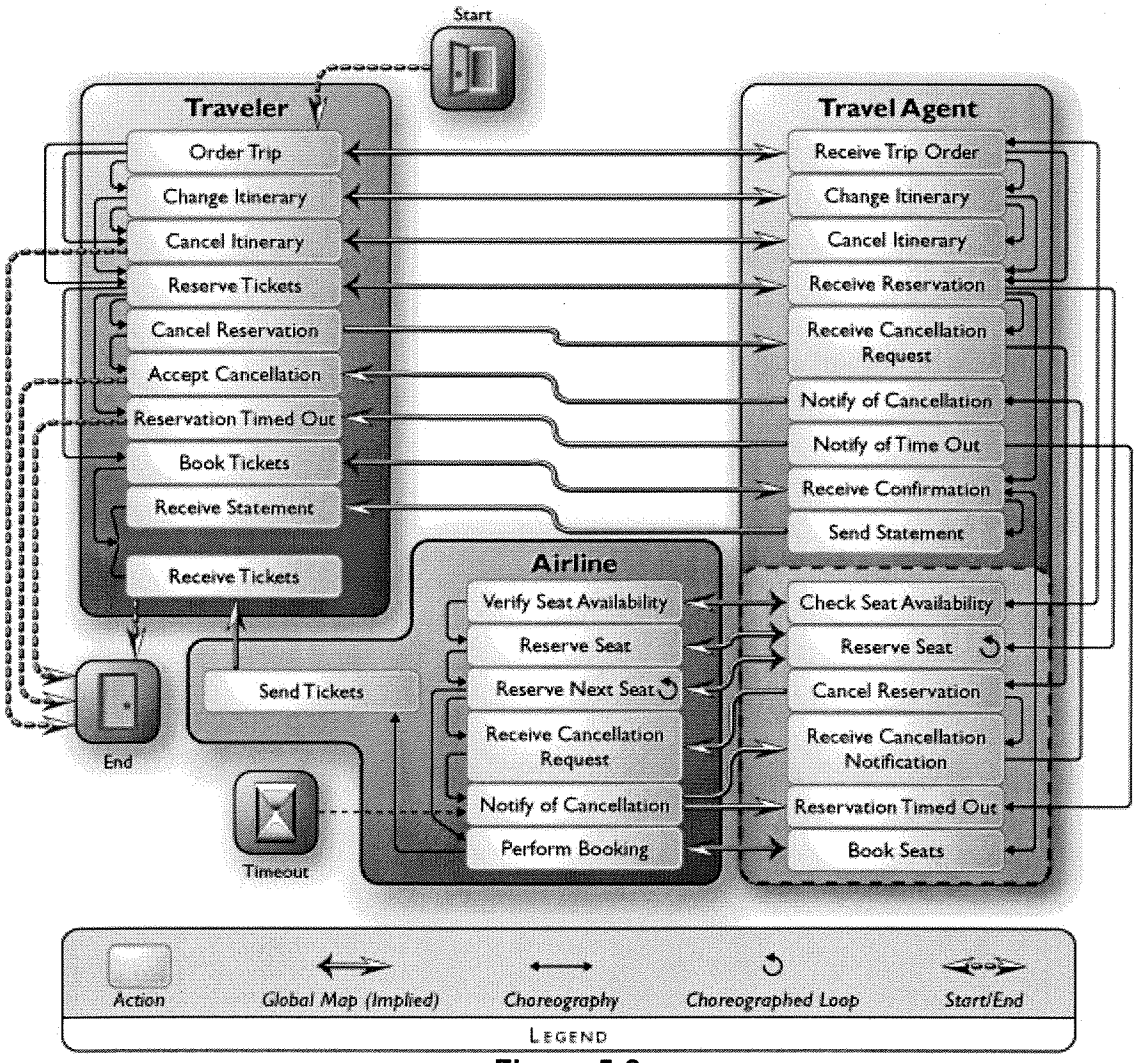


Figure 5-3

This picture shows:

- All the WSCI actions in each interface
- The choreography of actions within each interface, as described by each participant's WSCI interface
- The message flow from action to action

In the remaining part of this section, the details required to understand each participant's WSCI interface will be explained; the intent of the explanations has always been to describe how the WSCI modeling constructs have been used to

extend the simple scenario. Special attention has been given to present as many WSCI constructs as possible; therefore, different modeling techniques and constructs could have been used to describe the same 3 interfaces.

Note that the main top-level processes of each the participant interfaces are consistently named in the same way (*PlanAndBookTrip*). This is not enforced anywhere by WSCI; this convention has been chosen to explicitly show that all the interfaces refer to the same logical process (the Ticket Reservation process). In fact, each Web service may participate in more than one logical process and would expose a different interface for each one.

5.4 WSCI Interfaces

5.4.1 The Travel Agent Interface

5.4.1.1 Modeling details

- Once the Travel Agent receives an initial order from the Traveler (`ReceiveTripOrder` action), he needs to verify with the Airline Reservation System if seats are available for the desired trip. The verification is implemented via a nested process invoked within the `ReceiveTripOrder` action. This allows keeping the solicit-response semantic that is expected by the Traveler (the interaction of the Traveler should not be influenced by the way in which the Travel Agent actually communicates with other participants) and the corresponding request-response semantic for the Travel Agent.

  The `VerifySeats` nested process is described by means of a `while` activity whose condition is represented by the `openLegs` property; this property is evaluated against the internal implementation of the Travel Agent service.

  Note that the nested process is declared inside the `context` for the `sequence` activity.

- In order to model the possibility for the Traveler to change the initial itinerary, a nested processes (`ChangeItinerary`) with `instantiation=message` is declared in the context (note that a nested process with `instantiation=message` can occur at any time and any number of times inside the execution context).

  The `ChangeItinerary` nested process `calls` the `VerifySeat` nested process (this is a concrete example of how the `process` element allows reusability).

- If the Traveler decides to dismiss the trip request, an exception declared in the context captures the withdrawal. When the exception is caught, the context in which such exception is managed gets terminated; to terminate the `PlanAndBookTrip` process, the `fault` activity is used; without the use of

the `fault`, the execution would have resumed just after the activity set associated with the `context` declaring the exception handler.

- Note that the previous exception, as well as the `ChangeItinerary` nested process, are only available once the containing `context` is active; this ensures that the interface correctly describes that the Travel Agent needs to perform the `ReceiveTripOrder` action before being able to execute the nested process and/or process the exception.

- If the Traveler decides to confirm the itinerary, the `ReceiveReservation` action is executed. This action references the `itineraryCorrelation`; this correlation is, implicitly, instantiated by the previous action (`ReceiveTripOrder`) which actually provides the correlation identifier (itineraryID) to the Traveler. This identifier is the one that is used by the `itineraryCorrelation` and which makes sure that the proper context is retrieved.

- The `ReserveSeats` nested process is `called` from within the `ReceiveReservation` action and iterates though the selected legs using the `foreach` activity; at each iteration, a reservation of the related seat with the Airline Reservation System is performed (`ReserveSeat`).

  The `foreach` statement has been used (instead, for instance, of a `while` statement) since the number of iterations does not depend on any condition but only on the number of legs.

- The `ReserveFirstSeat` action outside of the `foreach` statement properly instantiates the `reservationCorrelation` correlation; this correlation, then, enables the Travel Agent to properly correlate subsequent messages from the Traveler (e.g. when booking the tickets) or the Airline Reservation System (e.g. when the reservation times out) to the conversation.

  Note that, within the `ReceiveReservation` action, the `itineraryCorrelation` is referenced; the `reservationCorrelation` is instantiated by means of the `ReserveFirstSeat` action in the called nested process.

- Once the reservation has been done, the Traveler could finally confirm the trip or change her mind and cancel the reservation. It could also happen that the reservation was hold for too long by the Airline Reservation System and, thus, it could be no more valid.

  To express these semantics, the behavior of the Travel Agent has been modeled via the use of an Exception Handler.

  - In case a timeout notification from the Airline Reservation System is received, a notification is sent to the Traveler and a `fault` activity

ensures that the process is terminated.

- In case the Traveler changes her mind and dismisses her reservation, the Travel Agent notifies the Airline Reservation System (in an asynchronous way, note the use of two separate actions involving the Travel Agent and the Airline Reservation System) and the Traveler is positively notified of the acceptance of her withdrawal. The execution of the `fault` activity ensures that the process is terminated.

- Once the Traveler confirms the reservation and such reservation is properly taken care by the Airline Reservation System (via the execution of the `BookSeats` nested process within the `ReceiveConfirmation` action), the Travel Agent bills the Traveler only in case the Credit Card details are accurate. This has been modeled by means of an `until` activity iterating over the `paymentOK` condition; the Travel Agents loops until he gets a confirmation containing accurate Credit Card details.

  `PaymentOK` is a condition internally managed by the Web service implementation; its value is `true` whenever the Credit Card of the Traveler is correctly debited.

- The interface exhibited by the Travel Agent uses two different correlations: `itineraryCorrelation` and `reservationCorrelation`. Collectively, the two correlations identify the same conversation involving the Travel Agent with both the Traveler and the Airline Reservation System.

5.4.1.2 The WSCI interface

```
<?xml version = "1.0" ?>
<wsdl:definitions   name = "TravelAgent"
   targetNamespace = "http://example.com/consumer/travelagent"
        xmlns = "http://www.w3.org/2002/07/wsci10"
        xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
        xmlns:tns = "http://example.com/consumer/travelagent"
        xmlns:defs = "http://example.com/consumer/definitions">

   <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/messages.wsdl" />
   <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/definitions.wsci" />

<!-- ************************************************************* -->
<!-- *************TRAVEL AGENT INTERFACE ************************** -->
<!-- ************************************************************* -->

<interface name = "TravelAgent">
   <documentation>
   The interface models the behavior of the travel agent service with
   respect to all of the other parties involved in the overall process.
   </documentation>
```

```
<process name = "PlanAndBookTrip" instantiation = "message">
  <sequence>
     <context>

       <process name = "VerifySeats" instantiation = "other">
         <while>
            <condition>tns:openLegs</condition>
            <action   name = "CheckSeatAvailability"
                 role = "tns:travelAgent"
                 operation = "tns:TAtoAirline/CheckSeatAvailability">
            </action>
         </while>
       </process>

       <process name = "ReserveSeats" instantiation = "other">
         <action   name = "ReserveFirstSeat"
                   role = "tns:travelAgent"
                   operation = "tns:TAtoAirline/ReserveSeat" >
            <correlate correlation = "defs:reservationCorrelation"
                       instantiation = "true" />
         </action>
         <foreach select="//proposedItinerary/leg[position()>1]">
            <action   name = "ReserveSeat"
                      role = "tns:travelAgent"
                      operation = "tns:TAtoAirline/ReserveSeat" />
         </foreach>
       </process>

       <process name = "ChangeItinerary"
                instantiation = "message">
         <action   name = "ChangeItinerary"
                   role = "tns:travelAgent"
                   operation = "tns:TAtoTraveler/ChangeItinerary">
            <correlate correlation="defs:itineraryCorrelation"/>
            <call process = "VerifySeats"/>
         </action>
       </process>

       <exception>
         <onMessage>
            <action   name = "CancelItinerary"
                      role = "tns:travelAgent"
                      operation = "tns:TAtoTraveler/CancelItinerary">
              <correlate correlation="defs:itineraryCorrelation"/>
            </action>
            <fault code = "tns:itineraryCancelled"/>
         </onMessage>
       </exception>

     </context>

     <action   name = "ReceiveTripOrder"
            role = "tns:travelAgent"
            operation = "tns:TAtoTraveler/OrderTrip">
       <call process = "VerifySeats"/>
```

```
    </action>

    <action   name = "ReceiveReservation"
          role = "tns:travelAgent"
          operation = "tns:TAtoTraveler/ReserveTickets">
      <correlate correlation = "defs:itineraryCorrelation" />
      <call process = "ReserveSeats"/>
    </action>
  </sequence>

  <until>
    <condition>tns:paymentOK</condition>
    <documentation>
    We catch the non-standard behavior meaning the reception of
    timeout and cancellation.  If this is an exception, the
    repeat-until is terminated and the sequence resumes outside
    the repeat-until
    </documentation>

    <context>
      <process name = "BookSeats" instantiation = "other">
        <action   name = "bookSeats"
                  role = "tns:travelAgent"
                  operation = "tns:TAtoAirline/bookSeats">
        </action>
      </process>

      <exception>
        <onMessage>
          <action   name = "ReservationTimedOut"
               role = "tns:travelAgent"
               operation = "tns:TAtoAirline/AcceptCancellation">
            <correlate
                 correlation = "defs:reservationCorrelation"/>
          </action>

          <action   name = "NotifyOfTimeout"
            role = "tns:travelAgent"
            operation = "tns:TAtoTraveler/NotifyOfCancellation"/>
                          <fault code = "tns:reservationTimedOut"/>
        </onMessage>

        <onMessage>
          <action name = "ReceiveCancellationRequest"
                  role = "tns:travelAgent"
                  operation = "tns:TAtoTraveler/CancelReservation">
            <correlate correlation="defs:reservationCorrelation"/>
          </action>

          <action name = "CancelReservation"
            role = "tns:travelAgent"
            operation = "tns:TAtoAirline/RequestCancellation"/>

          <action name = "ReceiveCancellationNotification"
               role = "tns:travelAgent"
               operation = "tns:TAtoAirline/AcceptCancellation">
```

```
                    <correlate
                         correlation = "defs:reservationCorrelation"/>
                  </action>

                  <action name = "NotifyOfCancellation"
                    role = "tns:travelAgent"
                    operation="tns:TAtoTraveler/NotifyOfCancellation"/>

                  <fault code = "tns:reservationCancelled"/>
                </onMessage>
              </exception>
            </context>

            <action   name = "ReceiveConfirmation"
                  role = "tns:travelAgent"
                  operation = "tns:TAtoTraveler/bookTickets">
              <correlate correlation="defs:reservationCorrelation"/>
              <call process = "BookSeats" />
            </action>
          </until>

          <action   name = "SendStatement"
                role = "tns:travelAgent"
                operation = "tns:TAtoTraveler/SendStatement"/>

        </process>
      </interface>

      <!-- ************************************************************* -->
      <!-- *************TRAVEL AGENT PORT TYPES ************************* -->
      <!-- ************************************************************* -->

        <wsdl:portType name = "TAtoTraveler">
          <wsdl:documentation>
            This port type contains operations which the Travel Agent
            service uses to connect with the Traveler service
          </wsdl:documentation>

          <wsdl:operation name = "OrderTrip">
            <wsdl:input message = "defs:tripOrderRequest"/>
            <wsdl:output message = "defs:tripOrderAcknowledgement"/>
          </wsdl:operation>

          <wsdl:operation name = "ChangeItinerary">
            <wsdl:input message = "defs:changeItineraryRequest"/>
            <wsdl:output message = "defs:changeItineraryConfirmation"/>
          </wsdl:operation>

          <wsdl:operation name = "CancelItinerary">
            <wsdl:input message = "defs:cancelItineraryRequest"/>
            <wsdl:output message = "defs:cancelItineraryConfirmation"/>
          </wsdl:operation>

          <wsdl:operation name = "ReserveTickets">
            <wsdl:input message = "defs:reservationRequest"/>
            <wsdl:output message = "defs:reservationConfirmation"/>
```

```
        </wsdl:operation>

        <wsdl:operation name="CancelReservation">
           <wsdl:input message="defs:reservationCancellationRequest"/>
        </wsdl:operation>

        <wsdl:operation name = "NotifyOfCancellation">
           <wsdl:output message = "defs:reservationCancellationResponse"/>
        </wsdl:operation>

        <wsdl:operation name = "bookTickets">
           <wsdl:input message = "defs:bookingRequest"/>
           <wsdl:output message = "defs:bookingConfirmation"/>
        </wsdl:operation>

        <wsdl:operation name = "SendStatement">
           <wsdl:output message = "defs:statement"/>
        </wsdl:operation>
      </wsdl:portType>

      <wsdl:portType name = "TAtoAirline">
        <wsdl:documentation>
           This port type contains operations which the Travel Agent
           service uses to connect  with the Airline service
        </wsdl:documentation>

        <wsdl:operation name = "CheckSeatAvailability">
           <wsdl:output message = "defs:seatAvailabilityCheck"/>
           <wsdl:input message = "defs:seatAvailability"/>
        </wsdl:operation>

        <wsdl:operation name = "ReserveSeat">
           <wsdl:output message = "defs:seatReservationRequest"/>
           <wsdl:input message = "defs:seatReservationConfirmation"/>
        </wsdl:operation>

        <wsdl:operation name = "RequestCancellation">
           <wsdl:output message = "defs:reservationCancellationRequest"/>
        </wsdl:operation>

        <wsdl:operation name = "AcceptCancellation">
           <wsdl:input message = "defs:reservationCancellationResponse"/>
        </wsdl:operation>

        <wsdl:operation name = "bookSeats">
           <wsdl:output message = "defs:ticketOrderRequest"/>
           <wsdl:input message = "defs:ticketOrderConfirmation"/>
        </wsdl:operation>
      </wsdl:portType>
    </wsdl:definitions>
```

5.4.2 The Traveler Interface

5.4.2.1 Modeling details

The Traveler interface has not been modeled in the simple example.

- The main, top-level process describing the behavior of the Traveler, is declared with the `instantiation=other` attribute. This describes the fact that the Traveler is actually the one who starts the message exchange associated with the logical process and that the start does not correspond to the receipt of any message described by the logical process.

- After ordering the trip (`OrderTrip` action), the Traveler can reserve the tickets for such trip (`ReserveTickets`). She could, also, change the chosen itinerary (`ChangeItinerary`) or withdraw her request (`CancelItinerary`) at any time; this may be because the Travel Agent was not able to properly select the desired itinerary or, simply, because the Traveler changed her mind.

  The activity of changing the itinerary is modeled by using a nested process (`ChangeItinerary`) defined with the `instantiation=other` attribute to signify that it is not automatically triggered by the reception of a specific message. A nested process can be started any time and any number of times within the context in which it is defined.

  The activity of withdrawing the request has been modeled in a `switch` statement since it can only be performed once; the `fault` statement explicitly terminates the process.

  Also note that the `switch` activity is enclosed in a `sequence` in order to constrain the usage of the nested process to the specific context in which the `switch` is performed.

- The Traveler can, then, book the tickets (`BookTickets`) or withdraw the reservation (`CancelReservation`) or be notified that the Airline Reservation System actually held the initial reservation for too long a period.

  The modeling of such situation is done by means of a new `context` defined in a new `sequence` activity; this new context defines an exception handler that manages the reception of the timeout notification from the Travel Agent Web service. The exception handler, once activated, terminates the execution of the top-level process because of the explicit use of the `fault` activity.

- The `BookTickets` action instantiates the `bookingCorrelation` which is, later, used by the Airline Reservation System and by the Travel Agent to ensure that the tickets and the statement are properly received by the Traveler.

  The correlation contains the `bookingID` property that is conveyed by the response from the Travel Agent. The Travel Agent himself receives the `bookingID` from the Airline; this ensures that both the Travel Agent and the Airline Reservation System know how to fulfill the correlation requirements

exhibited by the Traveler interface.

- Finally, the Traveler will receive the statement from the Travel Agent (`ReceiveStatement`) as well as the tickets from the Airline Reservation System (`ReceiveTickets`). Since these two actions do not need to be performed in any specific order, the `all` activity has been used.

  Note that the `bookingCorrelation` is used in both cases.

5.4.2.2 The WSCI interface

```
<?xml version = "1.0" ?>
<wsdl:definitions   name = "Traveler"
   targetNamespace = "http://example.com/consumer/traveler"
        xmlns = "http://www.w3.org/2002/07/wsci10"
        xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
        xmlns:tns = "http://example.com/consumer/traveler"
        xmlns:defs = "http://example.com/consumer/definitions">

   <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/messages.wsdl" />
   <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/definitions.wsci" />

<!-- ************************************************************* -->
<!-- ************TRAVELER INTERFACE ***************************** -->
<!-- ************************************************************* -->

<interface name = "Traveler">
   <documentation>
   The interface models the behavior of the Traveler service with
   respect to all of the other parties involved in the overall process.
   </documentation>

   <process name = "PlanAndBookTrip" instantiation = "other">

     <action   name = "OrderTrip"
            role = "tns:traveler"
            operation = "tns:TravelerToTA/OrderTrip"/>

     <sequence>
        <context>
          <process name = "ChangeItinerary" instantiation = "other">
            <documentation>
               Allows the traveler to change the itinerary as often as
               she likes.
            </documentation>
            <action   name = "ChangeItinerary"
                      role = "tns:traveler"
                      operation = "tns:TravelerToTA/ChangeItinerary"/>
          </process>
        </context>
```

```
    <switch>
      <case>
        <condition>tns:cancelItinerary</condition>
          <action   name = "CancelItinerary"
                role = "tns:traveler"
                operation = "tns:TravelerToTA/CancelItinerary"/>
          <fault code = "tns:exit"/>
      </case>
      <default>
        <action   name = "ReserveTickets"
                role = "tns:traveler"
                operation = "tns:TravelerToTA/ReserveTickets">
          <correlate correlation="defs:reservationCorrelation"
                     instantiation= "true" />
        </action>
      </default>
    </switch>
</sequence>

<sequence>
    <documentation>
    The context of this sequence traps the timeout for the
    reservation. The Timeout would abort the whole process.
    Also, it allows the traveler to cancel the trip.
    </documentation>
    <context>
      <exception>
        <onMessage>
          <action name = "ReservationTimedOut"
                role = "tns:traveler"
                operation = "tns:TravelerToTA/AcceptCancellation">
            <correlate correlation="defs:reservationCorrelation"/>
          </action>
          <fault code = "tns:exit"/>
        </onMessage>
      </exception>
    </context>

    <switch>
      <case>
        <condition>tns:cancelReservation</condition>
          <action   name = "CancelReservation"
               role = "tns:traveler"
               operation = "tns:TravelerToTA/RequestCancellation"/>
          <action   name = "AcceptCancellation"
                role = "tns:traveler"
                operation = "tns:TravelerToTA/AcceptCancellation">
            <correlate
              correlation = "defs:reservationCorrelation" />
          </action>
          <fault code = "tns:exit"/>
      </case>
      <default>
        <action   name = "BookTickets"
                role = "tns:traveler"
                operation = "tns:TravelerToTA/BookTickets">
```

```
                <correlate correlation = "defs:bookingCorrelation"
                           instantiation = "true"/>
            </action>
          </default>
        </switch>
      </sequence>

      <all>
        <action   name = "ReceiveTickets"
                role = "tns:traveler"
                operation = "tns:TravelerToAirline/ReceiveTickets">
          <correlate correlation = "defs:bookingCorrelation"/>
        </action>

        <action   name = "ReceiveStatement"
                role = "tns:traveler"
                operation = "tns:TravelerToTA/ReceiveStatement">
          <correlate correlation = "defs:bookingCorrelation"/>
        </action>
      </all>
    </process>
</interface>

<!-- ************************************************************* -->
<!-- ************TRAVELER PORT TYPES ***************************** -->
<!-- ************************************************************* -->

    <wsdl:portType name = "TravelerToTA">
      <wsdl:documentation>
        This port type contains operations which the Traveler service
        uses to connect with the Travel Agent service
      </wsdl:documentation>

      <wsdl:operation name = "OrderTrip">
        <wsdl:output message = "defs:tripOrderRequest"/>
        <wsdl:input message = "defs:tripOrderAcknowledgement"/>
      </wsdl:operation>

      <wsdl:operation name = "ChangeItinerary">
        <wsdl:output message = "defs:changeItineraryRequest"/>
        <wsdl:input message = "defs:changeItineraryConfirmation"/>
      </wsdl:operation>

      <wsdl:operation name = "CancelItinerary">
        <wsdl:output message = "defs:cancelItineraryRequest"/>
        <wsdl:input message = "defs:cancelItineraryConfirmation"/>
      </wsdl:operation>

      <wsdl:operation name = "ReserveTickets">
        <wsdl:output message = "defs:reservationRequest"/>
        <wsdl:input message = "defs:reservationConfirmation"/>
      </wsdl:operation>

      <wsdl:operation name = "RequestCancellation">
        <wsdl:output message = "defs:ReservationCancellationRequest"/>
      </wsdl:operation>
```

```
      <wsdl:operation name = "AcceptCancellation">
         <wsdl:input message = "defs:ReservationCancellationResponse"/>
      </wsdl:operation>

      <wsdl:operation name = "BookTickets">
         <wsdl:output message = "defs:bookingRequest"/>
         <wsdl:input message = "defs:bookingConfirmation"/>
      </wsdl:operation>

      <wsdl:operation name = "ReceiveStatement">
         <wsdl:input message = "defs:statement"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name = "TravelerToAirline">
      <wsdl:documentation>
         This port type contains operations which the Traveler service
         uses to connect with the Airline service
      </wsdl:documentation>

      <wsdl:operation name = "ReceiveTickets">
         <wsdl:input message = "defs:tickets"/>
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

5.4.3 The Airline Interface

5.4.3.1 Modeling details

The Airline Reservation System interface has not been modeled in the simple example. It exhibits two different top level processes, both with the `instantiation=message` attribute:

- `VerifySeats` models the activities that are performed when the Travel Agent requests the verification that the planned itinerary is actually possible.
- `PlanAndBookTrip` models the activities that are performed to actually reserve the seats, to finalize the reservation and to deliver the tickets.

In detail, the `PlanAndBookTrip` process uses the following modeling techniques:

- Each seat is actually reserved separately (corresponding to the `foreach` activity in the Travel Agent interface).

  The `ReserveSeat` action outside of the `while` loop is the first action within the `PlanAndBookTrip` process; the `reservationID` for the trip reservation is created at this point by the Airline and is communicated to the Travel Agent; this means that the Airline is instantiating the `reservationCorrelation` for

itself.

The Travel Agent uses the `reservationID` in order for the Airline Reservation System to properly relate the incoming messages to the execution context.

Note that both the Travel Agent and the Airline have instantiated the `reservationCorrelation` in their own process.

- The condition driving the `while` activity is modeled by means of an expression which evaluates to false when the last leg within the trip is received.

- The actions previously described are included in a `sequence` activity which establishes a `context` defining an atomic transaction (`seatReservation`).

  This transaction informs the client of the Airline Reservation System (the Travel Agent Web service, in this case) that the multiple seat reservations are actually performed in a transactional way by the service; since the transaction is `atomic`, this means that either all seat reservations will be performed or the Airline Reservation System will automatically roll them back.

  The transaction also defines a compensation activity which, once executed, probably will withdraw the reservations for all the seats (details of this behavior are of no interest from an interface point of view) and will notify the Travel Agent about the withdrawal (`NotifyOfCancellation` action). This notification is the only observable behavior exhibited by the compensation activity itself.

- The behavior of the Airline Reservation System at this point is described by means of the *choice* activity. The *choice* describes how the Web service can:

  - Receive a withdrawal notification from the Travel Agent, in which case it will simply *compensate* the *seatReservation* transaction (the compensation notifies the Travel Agent Web service about the positive execution of the withdrawal).

  - Receive the order of performing the final booking (*PerformBooking*), in which case it will send the tickets to the Traveler (actually, the tickets will be physically delivered at the Airline counter at the departure airport; electronically, a message is sent to the Traveler interface to confirm the physical action).

- Note that the *choice* activity previously described is, itself, contained in a *sequence* activity. The reason for this is that the *sequence* activity defines a *context* where the "timeout" exception is handled.

The timeout exception handler asserts the following:

- The timeout is actually calculated starting from the moment in which the *ReserveSeat* action terminates (a timer probably is started after the first seat has been reserved). The duration of the timeout is described by means of the property `tns:expiryTime`.

- The timeout exception is handled by executing the `compensation` activity for the `ReserveSeat` transaction (which, as usual, notifies the Travel Agent Web service about the positive execution of the withdrawal)

5.4.3.2 The WSCI interface

```
<?xml version = "1.0" ?>
<wsdl:definitions   name = "Airline"
   targetNamespace = "http://example.com/consumer/airline"
        xmlns = "http://www.w3.org/2002/07/wsci10"
        xmlns:defs = "http://example.com/consumer/definitions"
        xmlns:tns = "http://example.com/consumer/airline"
        xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/" >

   <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/messages.wsdl" />
   <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/definitions.wsci" />

<!-- ************************************************************* -->
<!-- *************AIRLINE INTERFACE ****************************** -->
<!-- ************************************************************* -->

<interface name="Airline">
   <documentation>
     The interface models the behavior of the Airline service with
     Respect to all of the other parties involved in the overall
     process.
   </documentation>

   <process name="VerifySeats" instantiation="message">
     <action name="VerifySeatAvailability"
          role="tns:Airline"
          operation="tns:AirlineToTA/VerifySeatAvailability">
     </action>
   </process>

   <process name="PlanAndBookTrip" instantiation= "message" >
     <sequence>
        <context>
          <transaction   name = "seatReservation"
                       type = "atomic">
            <compensation>
               <action   name = "NotifyOfCancellation"
                 role = "tns:Airline"
```

```
                operation = "tns:AirlineToTA/NotifyOfCancellation"/>
            </compensation>
          </transaction>
        </context>

        <action name="ReserveSeat"
            role ="tns:Airline"
            operation="tns:AirlineToTA/ReserveSeat"/>

        <while name="ReserveSeats">
          <condition>defs:notLastSeat</condition>
            <action name="ReserveNextSeat"
                 role ="tns:Airline"
                 operation="tns:AirlineToTA/ReserveSeat">
               <correlate correlation = "defs:reservationCorrelation" />
            </action>
        </while>
    </sequence>

    <sequence>
        <context>
          <exception>
            <onTimeout property = "tns:expiryTime"
                       type = "duration"
                       reference="tns:ReserveSeats@end">
               <compensate    name = "CompensateReservation"
                           transaction = "seatReservation"/>
            </onTimeout>
          </exception>
        </context>

        <choice>
          <onMessage>
            <action name = "ReceiveCancellationRequest"
                       role ="tns:Airline"
                       operation="tns:AirlineToTA/CancelReservation">
               <correlate correlation="defs:reservationCorrelation"/>
            </action>

            <compensate    name = "CompensateReservation"
                        transaction = "seatReservation"/>
          </onMessage>

          <onMessage>
            <action name="PerformBooking"
                    role ="tns:Airline"
                    operation="tns:AirlineToTA/BookSeats">
               <correlate correlation="defs:reservationCorrelation"/>
            </action>

            <action name="SendTickets"
                    role ="tns:Airline"
                    operation="tns:AirlineToTraveler/SendTickets"/>
          </onMessage>
        </choice>
    </sequence>
```

```
   </process>
</interface>

<!-- ************************************************************* -->
<!-- *************AIRLINE PORT TYPES ****************************** -->
<!-- ************************************************************* -->

   <wsdl:portType name="AirlineToTA">
     <wsdl:operation name="VerifySeatAvailability">
        <wsdl:input message="defs:seatAvailabilityCheck"/>
        <wsdl:output message="defs:seatAvailability"/>
     </wsdl:operation>

     <wsdl:operation name="ReserveSeat">
        <wsdl:input message="defs:seatReservationRequest"/>
        <wsdl:output message="defs:seatReservationConfirmation"/>
     </wsdl:operation>

     <wsdl:operation name="CancelReservation">
        <wsdl:input message="defs:reservationCancellationRequest"/>
     </wsdl:operation>

     <wsdl:operation name = "NotifyOfCancellation">
        <wsdl:output message = "defs:reservationCancellationResponse"/>
     </wsdl:operation>

     <wsdl:operation name="BookSeats">
        <wsdl:input message="defs:ticketOrderRequest"/>
        <wsdl:output message="defs:ticketOrderConfirmation"/>
     </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name="AirlineToTraveler">
     <wsdl:operation name="SendTickets">
        <wsdl:output message="defs:tickets"/>
     </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

5.5 Correlations and Selectors definitions

```
<?xml version = "1.0" ?>
<wsdl:definitions   name = "WSCI_Definitions"
   targetNamespace = "http://example.com/consumer/definitions"
   xmlns:tns = "http://example.com/consumer/definitions"
   xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
   xmlns = "http://www.w3.org/2002/07/wsci10">

   <wsdl:import   namespace = "http://example.com/consumer/definitions"
               location = "http://example.com/messages.wsdl" />

<!-- ******************************************************** -->
<!-- ***************** SELECTORS ************************** -->
<!-- ******************************************************** -->
```

```
    <selector property = "tns:itineraryNo"
           type = "tns:itineraryID"
           xpath = "./text()" />

    <selector property = "tns:itineraryNo"
           type = "tns:itinerary"
           xpath = "./itineraryID/text()" />

    <selector property = "tns:itineraryNo"
           type = "tns:proposedItinerary"
           xpath = "./itineraryID/text()" />

    <selector property = "tns:reservationNo"
           type = "tns:reservationID"
           xpath = "./text()" />

    <selector property = "tns:bookingNo"
           type = "tns:bookingID"
           xpath = "./text()" />

    <selector property = "tns:notLastSeat"
           type = "tns:seatToReserve"
           xpath = "./notLast" />

<!-- ********************************************************** -->
<!-- ****************** CORRELATIONS ********************** -->
<!-- ********************************************************** -->

    <correlation name = "itineraryCorrelation"
                 property = "tns:itineraryNo">
      <documentation>
      Correlation based on an itinerary number.
      </documentation>
    </correlation>

    <correlation name = "reservationCorrelation"
                 property = "tns:reservationNo">
      <documentation>
      Correlation based on a reservation number.
      </documentation>
    </correlation>

    <correlation name = "bookingCorrelation"
                 property = "tns:bookingNo">
      <documentation>
      Correlation based on a booking number.
      </documentation>
    </correlation>
</wsdl:definitions>
```

5.6 Global Model

5.6.1 Modeling details

Here is a graphical representation of the WSCI Global Model.

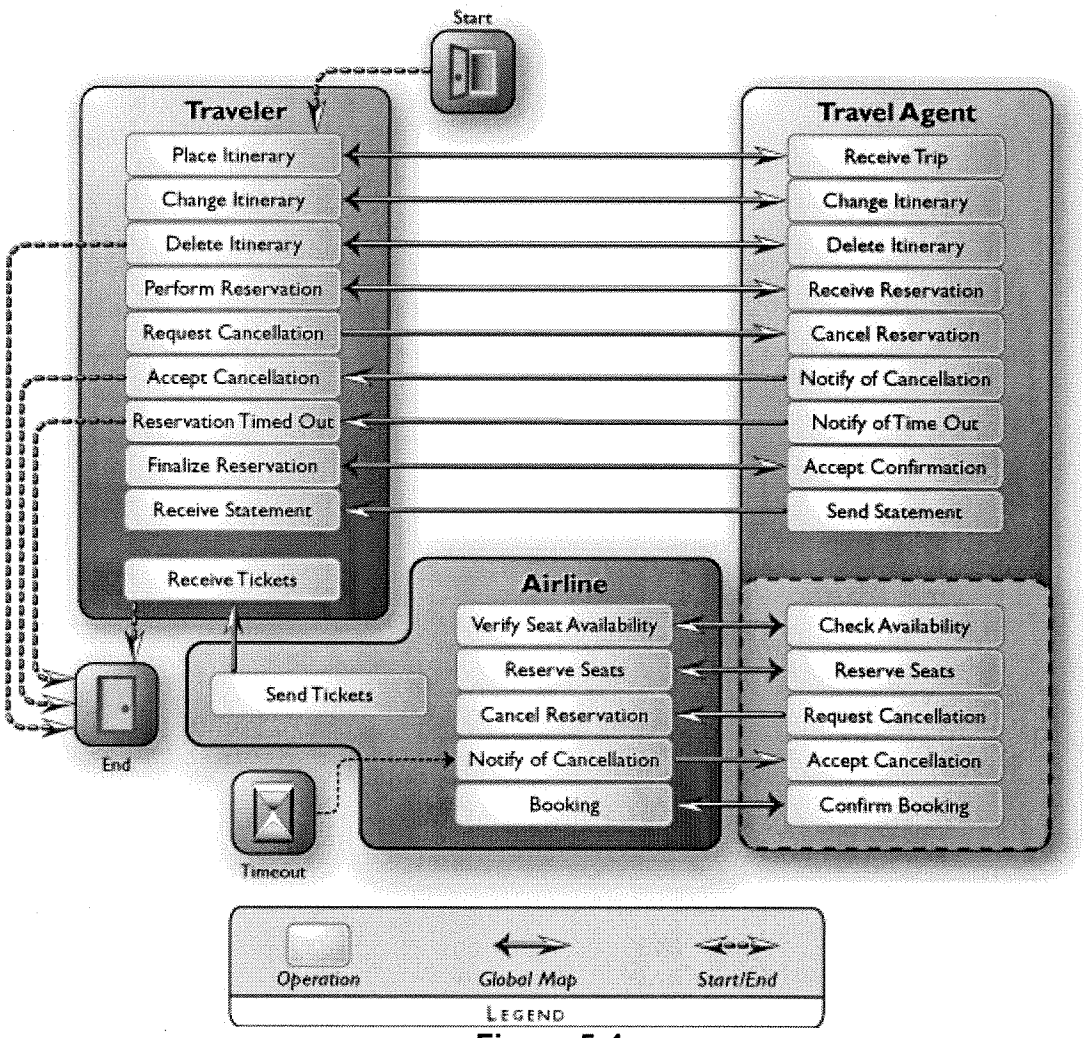


Figure 5-4

5.6.2 The WSCI global model

```
<?xml version = "1.0" ?>
<wsdl:definitions   name = "GlobalModel"
   targetNamespace = "http://example.com/consumer/models"
        xmlns = "http://www.w3.org/2002/07/wsci10"
        xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
        xmlns:tra = "http://example.com/consumer/traveler"
        xmlns:air = "http://example.com/consumer/airline"
        xmlns:ta = "http://example.com/consumer/travelagent">

   <wsdl:import   namespace = "http://example.com/consumer/traveler"
                location = "http://example.com/traveler.wsci" />
   <wsdl:import   namespace = "http://example.com/consumer/airline"
                location = "http://example.com/airline.wsci" />
```

```
    <wsdl:import   namespace = "http://example.com/consumer/travelagent"
                location = "http://example.com/travelagent.wsci" />

<model name = "AirlineTicketing">

     <interface ref = "air:Airline" />
     <interface ref = "tra:Traveler" />
     <interface ref = "ta:TravelAgent" />

<!-- Traveler / TravelAgent -->

     <connect operations = "tra:TravelerToTA/PlaceItinerary
                         ta:TAtoTraveler/ReceiveTrip" />

     <connect operations = "tra:TravelerToTA/ChangeItinerary
                         ta:TAtoTraveler/ChangeItinerary" />

     <connect operations = "tra:TravelerToTA/DeleteItinerary
                         ta:TAtoTraveler/DeleteItinerary" />

     <connect operations = "tra:TravelerToTA/PerformReservation
                         ta:TAtoTraveler/ReceiveReservation" />

     <connect operations = "tra:TravelerToTA/RequestCancellation
                         ta:TAtoTraveler/CancelReservation" />

     <connect operations = "tra:TravelerToTA/AcceptCancellation
                         ta:TAtoTraveler/NotifyOfCancellation" />

     <connect operations = "tra:TravelerToTA/FinalizeReservation
                         ta:TAtoTraveler/AcceptConfirmation" />

     <connect operations = "tra:TravelerToTA/ReceiveStatement
                         ta:TAtoTraveler/SendStatement" />

<!-- Traveler / Airline -->

     <connect operations = "tra:TravelerToAirline/ReceiveTickets
                         air:AirlineToTraveler/SendTickets"/>

<!-- Travel Agent / Airline -->

     <connect operations = "ta:TAtoAirline/CheckAvailability
                         air:AirlineToTA/VerifySeatAvailability"/>

     <connect operations = "ta:TAtoAirline/ReserveSeats
                         air:AirlineToTA/ReserveSeats"/>

     <connect operations = "ta:TAtoAirline/RequestCancellation
                         air:AirlineToTA/CancelReservation" />

     <connect operations = "ta:TAtoAirline/AcceptCancellation
                         air:AirlineToTA/NotifyOfCancellation" />

     <connect operations = "ta:TAtoAirline/ConfirmBooking
                         air:AirlineToTA/Booking"/>
```

```
   </model>
</wsdl:definitions>
```

5.7 WSDL Types and Messages

This file describes the WSDL abstract messages that the participants in the Airline Ticketing Example exchange among themselves, and the types required to define these messages.

```
<?xml version = "1.0" ?>
<definitions   name = "TypesAndMessages"
   targetNamespace = "http://example.com/consumer/definitions"
   xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema"
   xmlns:xsd1 = "http://example.com/consumer/definitions"
   xmlns = "http://schemas.xmlsoap.org/wsdl/">

   <types>
     <schema   xmlns = "http://www.w3.org/2000/10/XMLSchema">

<!-- ************************************************************* -->
<!-- ****************** SIMPLE TYPES ************************** -->
<!-- ************************************************************* -->

        <simpleType name = "itineraryID">
        <restriction base = "string"/>
        </simpleType>
        <simpleType name = "reservationID">
        <restriction base = "string"/>
        </simpleType>
        <simpleType name = "bookingID">
        <restriction base = "string"/>
        </simpleType>
        <simpleType name = "ticket">
        <restriction base = "string"/>
        </simpleType>

        <simpleType name = "status">
        <enumeration value = "created"/>
        <enumeration value = "changed"/>
        <enumeration value = "cancelled"/>
        <enumeration value = "reserved"/>
        <enumeration value = "timedOut"/>
        </simpleType>

<!-- ************************************************************* -->
<!-- ****************** COMPLEX TYPES ************************** -->
<!-- ************************************************************* -->
        <complexType name = "baseAddress">
        <sequence>
          <element name = "street" type = "string" />
          <element name = "city" type = "string" />
          <element name = "state" type = "string" />
          <element name = "ZIP" type = "string" />
        </sequence>
```

```
</complexType>

<complexType name = "address">
<complexContent>
  <extension base = "xsd1:baseAddress">
    <sequence>
       <element name = "email" type = "string" />
       <element name = "phone" type = "string" />
    </sequence>
  </extension>
</complexContent>
</complexType>

<complexType name = "person">
<sequence>
  <element name = "name" type = "string" />
  <element name = "address" type = "xsd1:address" />
</sequence>
</complexType>

<complexType name = "traveler">
<complexContent>
  <extension base = "xsd1:person">
    <element name = "travelerID" type = "string" />
  </extension>
</complexContent>
</complexType>

<complexType name = "flight">
<sequence>
  <element name = "startTime" type = "timeInstant" />
  <element name = "startAirport" type = "string" />
  <element name = "destinationTime" type = "timeInstant" />
  <element name = "destinationAirport" type = "string" />
  <element name = "carrier" type = "string" />
  <element name = "cost" type = "float" />
</sequence>
</complexType>

<complexType name = "leg">
<complexContent>
  <extension base = "xsd1:flight">
    <sequence>
       <element name = "availability" type = "boolean" />
    </sequence>
  </extension>
</complexContent>
</complexType>

<complexType name = "itinerary">
<sequence>
  <element name = "itineraryID" type = "xsd1:itineraryID"/>
  <element name = "leg"
           type = "xsd1:leg" maxOccurs = "unbounded"/>
  <element name = "comments" type = "string" />
</sequence>
```

```
</complexType>

<complexType name = "proposedItinerary">
<complexContent>
  <extension base = "xsd1:itinerary">
    <sequence>
       <element name = "totalCost" type = "float" />
       <element name = "validityDeadline"
                type = "timeInstant" />
    </sequence>
  </extension>
</complexContent>
</complexType>

<complexType name = "trip">
<sequence>
  <element name = "startDate" type = "date" />
  <element name = "startCity" type = "string" />
  <element name = "arrivalDate" type = "date" />
  <element name = "destinationAirport" type = "string" />
  <element name = "seatPreferences"
            type = "xsd1:seatPreferences" />
  <element name = "numberOfSeats"
            type = "nonNegativeInteger"/>
  <element name = "preferredCarrier" type = "string" />
  <element name = "comments" type = "string" />
</sequence>
</complexType>

<complexType name = "CCInfo">
<sequence>
  <element name = "number" type = "string" />
  <element name = "issuer" type = "string" />
  <element name = "expiryDate" type = "month" />
</sequence>
</complexType>

<complexType name = "statement">
<sequence>
  <element name = "bookingID" type = "xsd1:ID" />
  <element name = "creditCard" type = "xsd1:CCInfo" />
  <element name = "date" type = "date" />
  <element name = "amount" type = "float"/>
  <element name = "transactionID" type = "xsd1:ID"/>
</sequence>
</complexType>

<complexType name = "seatPreferences">
<sequence>
  <element name = "class" type = "string"/>
  <element name = "smoking" type = "boolean"/>
  <element name = "window" type = "boolean"/>
</sequence>
</complexType>

<complexType name = "seatDetails">
```

```
            <sequence>
              <element name = "flight" type = "xsd1:flight"/>
              <element name = "preferences"
                       type = "xsd1:seatPreferences"/>
            </sequence>
            </complexType>

            <complexType name = "seatAvailability">
            <complexContent>
              <extension base = "xsd1:seatDetails">
                <sequence>
                   <element name = "availability" type = "boolean"/>
                </sequence>
              </extension>
            </complexContent>
            </complexType>

            <complexType name = "seatToReserve">
            <complexContent>
              <extension base = "xsd1:seatDetails">
                <sequence>
                   <element name = "notLast" type = "boolean"/>
                </sequence>
              </extension>
            </complexContent>
            </complexType>

            <complexType name = "tickets">
            <sequence>
              <element name = "ticket" type = "xsd1:ticket"
                       minOccurs = "0"/>
            </sequence>
            </complexType>
          </schema>
        </types>

<!-- ************************************************************ -->
<!-- ****************** MESSAGES ******************************** -->
<!-- ************************************************************ -->

<!-- between Travel Agent and Traveler -->

        <message name = "tripOrderRequest">
          <documentation>
             Contains the trip information that the traveler sends to the
             travel agent.
          </documentation>
          <part name = "traveler" type = "xsd1:traveler"/>
          <part name = "trip" type = "xsd1:trip"/>
        </message>

        <message name = "tripOrderAcknowledgement">
          <documentation>
             Contains the itinerary that the travel agent proposes to the
             traveler. Itinerary Identification
          </documentation>
```

```
    <part name = "proposedItinerary" type = "xsd1:proposedItinerary"/>
</message>

<message name = "changeItineraryRequest">
    <documentation>
        Contains the itinerary to be changed
    </documentation>
    <part   name = "itinerary" type = "xsd1:itinerary"/>
</message>

<message name = "changeItineraryConfirmation">
    <documentation>
        Contains the itinerary changed by the travel agent.
    </documentation>
    <part name = "proposedItinerary" type = "xsd1:proposedItinerary"/>
</message>

<message name = "cancelItineraryRequest">
    <documentation>
        Contains the itinerary to be deleted
    </documentation>
    <part   name = "itineraryID" type = "xsd1:itineraryID"/>
</message>

<message name = "cancelItineraryConfirmation">
    <documentation>
        Contains the status of the deletion of an itinerary that the
        traveler has requested from the travel agent.
    </documentation>
    <part   name = "itineraryID" type = "xsd1:itineraryID"/>
    <part   name = "status" type = "xsd1:status"/>
</message>

<message name = "reservationRequest">
    <documentation>
        Contains a request from the traveler to the travel agent
        to reserve the seats for all the legs of an itinerary,
        plus the traveler's credit card information.
    </documentation>
    <part   name = "itineraryID" type = "xsd1:itineraryID"/>
    <part   name = "CCInfo" type = "xsd1:CCInfo"/>
</message>

<message name = "reservationConfirmation">
    <documentation>
        Contains the status of a reservation request that is sent from
        the travel agent to the traveler
    </documentation>
    <part   name = "reservationID" type = "xsd1:reservationID"/>
    <part   name = "status" type = "xsd1:status"/>
</message>

<message name = "reservationTimeout">
    <documentation>
        Contains the signal that the reservation of seats has timed out.
    </documentation>
```

```
    <part   name = "reservationID" type = "xsd1:reservationID"/>
    <part   name = "status" type = "xsd1:status"/>
  </message>

  <message name = "bookingRequest">
    <documentation>
       Contains the signal that the traveler confirms a reservation for
       seats
    </documentation>
    <part   name = "reservationID" type = "xsd1:reservationID"/>
  </message>

  <message name = "bookingConfirmation">
    <documentation>
       Contains the signal that the travel agent has reserved seats
    </documentation>
    <part   name = "bookingID" type = "xsd1:bookingID"/>
    <part   name = "status" type = "xsd1:status"/>
  </message>

  <message name = "statement">
    <documentation>
       Contains the statement send from the travel agent to a client
    </documentation>
    <part   name = "bookingID" type = "xsd1:bookingID"/>
    <part   name = "body" type = "xsd1:statement"/>
  </message>

<!-- between Travel Agent and Airline -->

  <message name = "seatAvailabilityCheck">
    <documentation>
       Contains the details of an airplane seat for which the
       availability is to be checked.
    </documentation>
    <part   name = "seatDetails" type = "xsd1:seatDetails"/>
  </message>

  <message name = "seatAvailability">
    <documentation>
       Contains the confirmation or rejection of the availability of a
       seat.
    </documentation>
    <part   name = "seatAvailability" type = "xsd1:seatAvailability"/>
  </message>

  <message name = "seatReservationRequest">
    <documentation>
       Contains the details of an airplane seat that is to be reserved.
    </documentation>
    <part   name = "reservationID" type = "xsd1:reservationID"/>
    <part   name = "seatToReserve" type = "xsd1:seatToReserve"/>
  </message>

  <message name = "seatReservationConfirmation">
    <documentation>
```

```
            Contains the confirmation or rejection regarding the reservation
            of a seat.
          </documentation>
          <part   name = "reservationID" type = "xsd1:reservationID"/>
          <part   name = "status" type = "xsd1:status"/>
        </message>

        <message name = "ticketOrderRequest">
          <documentation>
            Contains the confirmation of an airline client that it books
            some seats on behalf of some recipient
          </documentation>
          <part   name = "reservationID" type = "xsd1:reservationID"/>
          <part   name = "recipient"    type = "xsd1:person"/>
        </message>

        <message name = "ticketOrderConfirmation">
          <documentation>
            Contains the signal that the airline sends to a client that
            indicates the booking success
          </documentation>
          <part   name = "bookingID" type = "xsd1:bookingID"/>
          <part   name = "status" type = "xsd1:status"/>
        </message>

    <!-- between Traveler and Airline -->

        <message name = "tickets">
          <documentation>
            Contains the e-tickets sent from the airline serviceto the
            traveler service.
          </documentation>
          <part   name = "bookingID" element = "xsd1:bookingID"/>
          <part   name = "tickets"    element = "xsd1:tickets"/>
        </message>

    <!-- Common -->

        <message name = "reservationCancellationRequest>"
          <documentation>
            Contains the signal that the sender requests the receiver to
            cancel a reservation
          </documentation>
          <part   name = "reservationID" type = "xsd1:reservationID"/>
        </message>

        <message name = "reservationCancellationResponse>"
          <documentation>
            Contains the status of a reservation cancellation request
            that has been issued previously
          </documentation>
          <part   name = "reservationID" type = "xsd1:reservationID"/>
          <part   name = "status" type = "xsd1:status"/>
        </message>
    </definitions>
```

6. Dynamic Participation

The interface definition describes the behavioral aspects of a service. The previous sections introduce concepts, such as message choreography, message correlation, transactions and exception handling, that are important to understand how a service behaves in the context of a given message exchange. This section introduces another behavioral aspect: the ability of identifying the target service when performing an operation which begins with sending a message by means of information that is conveyed within the message exchange.

A message exchange, as described by a WSCI interface, is based on the assumption that each Web service will actually play a (named) role in the exchange. When modeling, roles are enough to describe the choreography. At runtime, though, the identity of the target service when performing an operation that begins with sending a message needs to be fully defined for such operation to take place.

Sometimes the identity of the target service does not depend on any information described by the WSCI interface; in this case, the action that performs an operation that begins with sending a message does not need to be further qualified.

In other cases, though, the identity of the target service is dynamically selected based on some criteria that is known at runtime and that depends on information described by the WSCI interface, such as message parts. This is equivalent to consider that the set of services participating in the exchange can vary during the execution of the exchange itself. The ability to model how services can be dynamically identified during the message exchange via information conveyed by the exchange itself is called dynamic participation: the service's identifier (its address or communication end point) can be transmitted as data in a message and, optionally, a complex lookup procedure could also be specified.

> In the example presented in Section 5, the Airline Reservation System sends an eTicket to the traveler; either the Airline knows the identity of the traveler (in which case the relevant action does not need further qualification) or it retrieves the identity from a parts of messages exchanged with the Travel Agent. In the latter case, a locator can be used.

The `locate` and `locator` elements are the WSCI constructs that realize the concept of dynamic participation. These two elements are included as a normative extension to the specification. A new namespace (`http://www.w3.org/2002/07/wsci10/locate` namespace) has been introduced to explicitly separate the WSCI core elements from the normative extension that

includes the definitions of the `locate` and `locator` elements. Not all WSCI implementations would necessarily support dynamic participation.

> **Note:** A WSCI implementation that ignores the `locate` and `locator` elements, but supports all other WSCI elements, is considered WSCI-compliant. An implementation that understands the `locate` and `locator` elements is considered compliant to the extended behavior.

6.1 Locate

The `locate` element is used to identify a service against which the action will be performed (target service). It is defined as an optional extension element of the action element. At most one `locate` element can be specified for an action. The locate element identifies the service to which the message will be sent. An action that is receiving a message does not require any locate element since it is performed by the service whose interface is under consideration.

The `locate` element specifies which properties and/or mechanism are used to identify a particular target service. In some situations (replay-to address and redirection, for instance) the URI of the target service can be simply specified via a property. In other situations, more complex locator mechanisms (e.g. UDDI lookup) might be used to resolve the service.

By knowing the properties used to locate the service it is possible to understand how, changing the value of these properties, the behavior of the corresponding services is affected.

The syntax of the `locate` element is:

```
<locate
   property = list of QName
   locator = QName/>
```

The `property` attribute lists all properties used to identify a particular service. A property is referenced using a qualified name. The `property` attribute is optional.

The `locator` attribute references a `locator` element (which, in turn, describes the mechanism that, by means of the properties listed in the `locate` element, is able to identify the service). The `locator` is referenced using a qualified name. Thus, it is possible to use locators defined in a namespace other than the namespace used for the interface definition. The `locator` attribute is optional.

A `locate` element must specify at least one of these two attributes. If the `locator` attribute is omitted exactly one property must be specified which value is a URI of the service. If the `property` attribute is omitted the locate element must specify the locator attribute. In this case the locator element is not dependent on any property and the target service is assumed to be always the same (the service is specified by the locator definition). If at least one property is given and a locator attribute is specified, it is assumed that the locator mechanism uses the property list to resolve the service. The locator mechanism is required if the service is not identified by a URI.

6.2 Locator

The `locator` element specifies a mechanism used to identify a particular service.

The syntax of the `locator` element is:

```
<locator
    name = NCName
    property = list of QName
    {extension attribute}>
    Content: (documentation?,
    {extension element})
</locator>
```

The `name` attribute provides a name for the locator. This attribute is mandatory. The name must be unique among names of locators defined in the same namespace.

The `property` attribute lists all properties used to resolve the service. This property list defines the list of formal parameters used by the locator mechanism. The property list in the locate element defines the actual parameters for that mechanism. The properties in the locator element are given values from the corresponding properties listed in the `locate` element, using the same order. A locator element may include the documentation element, which contains arbitrary text.

The definition of the mechanism used to identify a specific service is out of the scope of WSCI and can be addressed by existing and future specifications. For that reason the locator element is defined as an extensible element. A WSCI implementation is not required to understand how the locator mechanism (e.g. UDDI lookup) works. A locator element must include exactly one extension element and may include zero or more extension attributes that can be used to provide additional information for the purpose of locating services.

A locator definition is not limited to a specific interface definition. The `locator` element is a WSCI top-level element.

Example. This example extends the example given in Section 5 by introducing the

locate element. The final part of the airline ticket reservation process encompasses the booking of seats activity. The Traveler initiates this activity sending the reservation confirmation to the Travel Agent, which, after that, requests from the Airline Reservation System to book the seats. The notification of the successful booking is sent to Travel Agent and the confirmation of the issued tickets is sent to the Traveler.

The `interface` definition of the Airline Reservation System given in Section 5.4.3 does not identify the service to whom the confirmation of the issued tickets will be sent. Consequently, the Travel Agent cannot conclude, from the interface definition of the Airline Reservation System, that the message will be sent to the Traveler that initiated the booking of seats. The Traveler's name and email address are conveyed in the bookingRequest message sent to the Airline Reservation System by the Travel Agent. The Airline Reservation System does not previously know any specific traveler.

This example provides a modified version of the Airline interface. The `SendTickets` action uses the locate element to identify the Traveler to whom the message will be sent. The `locate` element uses the `replyToAddress` property which value is obtained from the incoming `bookingRequest` message. The accompanying selector definition is given below; the complex type `traveler` is defined in Section 5.7.

```
<?xml version = "1.0" ?>
<wsdl:definitions   name = "WSCI_Definitions"
   targetNamespace = "http://example.com/consumer/definitions"
   xmlns:tns = "http://example.com/consumer/definitions"
   xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
   xmlns = "http://www.w3.org/2002/07/wsci10">

   <wsdl:import   namespace = "http://example.com/consumer/definitions"
               location = "http://example.com/messages.wsdl" />

<!-- ************************************************************ -->
<!-- Definitions of selectors and correlations introduced in      -->
<!-- section 5.7 are omitted.                                     -->
<!-- ************************************************************ -->

   <selector property = "tns:replyToAddress"
          type = "tns:traveler"
          xpath = "./email/text()" />

</wsdl:definitions>
<?xml version = "1.0" ?>
<wsdl:definitions   name = "Airline"
   targetNamespace = "http://example.com/consumer/airline"
        xmlns = "http://www.w3.org/2002/07/wsci10"
        xmlns:defs = "http://example.com/consumer/definitions"
        xmlns:tns = "http://example.com/consumer/airline"
        xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsci1 = "http://www.w3.org/2002/07/wsci10/locate">
```

```
    <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/messages.wsdl" />
    <wsdl:import   namespace = "http://example.com/consumer/definitions"
                location = "http://example.com/definitions.wsci" />

<!-- ************************************************************* -->
<!-- *************AIRLINE INTERFACE ****************************** -->
<!-- ************************************************************* -->

<interface name="Airline">
   <process name="VerifySeats" instantiation="message">
     <action name="VerifySeatAvailability"
          role="tns:Airline"
          operation="tns:AirlineToTA/VerifySeatAvailability"/>
   </process>

   <process name="PlanAndBookTrip" instantiation= "message" >
     <sequence>
        <context>
          <transaction   name = "seatReservation"
                      type = "atomic">
            <compensation>
               <action   name = "NotifyOfCancellation"
                 role = "tns:Airline"
                 operation = "tns:AirlineToTA/NotifyOfCancellation"/>
            </compensation>
          </transaction>
        </context>

        <action name="ReserveSeat"
            role ="tns:Airline"
            operation="tns:AirlineToTA/ReserveSeat"/>

        <while name="ReserveSeats">
          <condition>defs:notLastSeat</condition>
          <action name="ReserveNextSeat"
               role ="tns:Airline"
               operation="tns:AirlineToTA/ReserveSeat">
          <correlate correlation = "defs:reservationCorrelation" />
          </action>
        </while>
     </sequence>

     <sequence>
        <context>
          <exception>
            <onTimeout property = "tns:expiryTime"
                      type = "duration"
                      reference="tns:ReserveSeats@end">
               <compensate    name = "CompensateReservation"
                           transaction = "seatReservation"/>
            </onTimeout>
          </exception>
        </context>

        <choice>
```

```
            <onMessage>
            <action name = "ReceiveCancellationRequest"
                    role ="tns:Airline"
                    operation="tns:AirlineToTA/CancelReservation">
                <correlate correlation="defs:reservationCorrelation"/>
              </action>

            <compensate   name = "CompensateReservation"
                        transaction = "seatReservation"/>
          </onMessage>

          <onMessage>
              <action name="PerformBooking"
                    role ="tns:Airline"
                    operation="tns:AirlineToTA/BookSeats">
               <correlate correlation="defs:reservationCorrelation"/>
            </action>

            <action name="SendTickets"
                    role ="tns:Airline"
                    operation="tns:AirlineToTraveler/SendTickets">
               <wscil:locate property = "defs:replyToAddress" />
              </action>
          </onMessage>
        </choice>
      </sequence>
    </process>
  </interface>
```

---

7. Future Work

WSCI is a work in progress; as such there areas, which we feel, will be expanded on as the specification matures.

7.1 Exclusion Groups

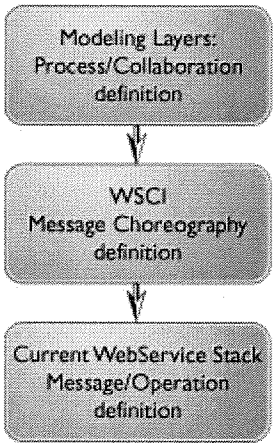
For a generic model we think that it might be useful to have a possibility to specify properties ("exclusion groups") indicating services that must be excluded from a set of services against the action can be performed. Our first thought was that these services should be specified using a single property whose value is a URI (but no restriction must be made). For example, exclusion groups can be used to specify that a service against the action performed is not the service used in a previous action (step). In this case, we assume that a locator has the name and instantiates property with the same name whose value is the service identifier (e.g. URI). The service used in a previous action (step) can be referenced using the name of the corresponding locator.

7.2 Web based Collaboration

At the time of submission of this standard the web services stack is in very early stages of evolution.

We view the ultimate goal of web services as enabling web-based collaboration.

As stated in the introduction, and as re-illustrated here, we see layers above WSCI defining the aspects of such collaborations, and WSCI in essence reifying the interface of each participant in such a collaboration.



As the collaboration layer(s) of the web services stack evolve to define aspects such as Contract, State, Role, and Rights and Obligations of Participants, WSCI will need to evolve in parallel to facilitate the mapping of WSCI elements to these anticipated elements of the Collaboration layer(s). For instance, we expect the current Role element evolving to map to a Role at the collaboration level, and the Process element to map to a Process element at the collaboration level. As these mappings get created, they may require syntax changes to either the WSCI specification and/or the Collaboration specification.

7.3 Global Model

In this version of the WSCI specification the global model consists of mappings of operations of one web service to operations of another web service. This choice was made to preserve the close relationship to the web service description languages, such as WSDL, where atomic web service operations are described. As the higher layers of the web service stack mature, we anticipate that the focus will be more on the collaboration aspects of the global model. This would mean a focus more on describing the interaction between two (or more) participants, and less on the lower level one sided operation definitions. Most likely future work will include a global mapping of WSCI actions to each other, and an evolution of the relationship between roles and actions.

With the current specification the global model assumes a pretty tight coupling between the WSCI action and the WSDL operation. This, to a degree, limits the

ability of a given WSDL operation to be re-used in multiple actions, as care has to be given to the unique naming of the actions and operations. Lifting the level of the global model up to the action level would be a first step towards a looser coupling and a greater re-usability of operations across multiple actions.

However, embarking on this work would be premature until the evolving collaboration models are more stable and well known.

---

8. Appendix

8.1 Known Issues

The current version of the WSCI specification and, specifically, the design of the relationship between WSCI and WSDL, is based on the W3C acknowledged submission of the WSDL specification. It is acknowledged that the WSDL specification may evolve in the future and, although the relationship between WSCI and WSDL is deliberately loosely coupled, each future change to the WSDL specification will be reviewed as to its potential change to WSCI.

8.2 References

[WSDL]
Web Services Description Language (WSDL) 1.1, Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, Editors. World Wide Web Consortium, 15 March 2001. WSDL 1.1 is available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[RFC 2119]
Key words for use in RFCs to Indicate Requirement Levels, S. Bradner. IETF, March 1997. RFC 2119 is available at http://www.normos.org/ietf/rfc/rfc2119.txt.

[RFC 2396]
Uniform Resource Identifiers (URI): Generic Syntax, T. Berners-Lee, R. Fielding, L. Masinter. IETF, August 1998. RFC 2396 is available at http://www.normos.org/ietf/rfc/rfc2396.txt

[XML Schema 1]
XML Schema Part 1: Structures, Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, Editors. World Wide Web Consortium, 2 May 2001. This version of the XML Schema Part 1 recommendation is available at http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/. The latest version of XML Schema Part 1 is available at http://www.w3.org/TR/xmlschema-1/.

[XML Schema 2]
XML Schema Part 2: Datatypes, Paul V. Biron, Ashok Malhotra, Editors. World Wide Web Consortium, 2 May 2001. This version of the XML Schema

Part 2 recommendation is available at http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/. The latest version of XML Schema Part 2 is available at http://www.w3.org/TR/xmlschema-2/.

[XPATH]
XML Path Language (XPath) Version 1.0, James Clark, Steve DeRose, Editors. World Wide Web Consortium, 16 November 1999. This version of XPath 1.0 recommendation is available at http://www.w3.org/TR/1999/REC-xpath-19991116. The latest version of XPath 1.0 is available at http://www.w3.org/TR/xpath.

8.3 Glossary

This glossary contains definitions of key terms used within this WSCI specification. The definitions are as seen from a WSCI perspective. Some of these terms may have different meaning in other environments.

Included in this glossary are terms that have important conceptual content for the understanding of this specification. Not included in this glossary are the names of more specific WSCI language constructs. For discussion of such language constructs, see Section 3.

This glossary is in alphabetic order. See Section 2 for an introduction to the concepts in logical order.

Action: An atomic activity that describes the manner in which a service performs a single atomic message exchange as defined, for instance, by a WSDL operation.

Activity: Any of the WSCI language constructs that describe simple or complex behavior. An activity is the basic building block for defining behavior within a process.

Atomic Activity: An activity definition that cannot be decomposed into further activities. It represents an atomic unit of work. There are two types of atomic activities: Action and Delay.

Activity Instance: An Activity being executed.

Activity Set: Defines a set of activities together with the context in which they are performed. An Activity Set is a modeling artifact mainly used to describe Complex Activities; it does not correspond directly to any WSCI language construct.

Choreography: Describes temporal and/or logical dependencies among Activities.

Collaboration: The act of a set of participants, for instance a set of Web services, working together in complex relationships to meet a common goal. Definition of

Collaboration requires the definition of roles, responsibilities, contracts, artifacts, state management and state transitions binding involving all the participants' involved parties. Definition of Collaboration is out of the scope of WSCI, but it is assumed that such a definition exists formally or informally.

Compensation: A transaction may declare a set of compensating activities that are to be executed when a successfully completed transaction needs to be undone. Compensation describes only the externally observable activities required to undo the transaction; it does not describe how the transaction is undone by the implementation.

Complex Activity: An Activity definition that is composed of other Activities. A complex activity defines a choreography for the activities of which it is composed.

Context: declares characteristics that affect the execution of a particular set of activities, in terms of local properties, local process definitions, transactional characteristics and exceptional behavior. Contexts can be nested to an arbitrary level.

Conversation: The message traffic associated with one execution of a choreography (or part thereof), as a service interacts with other services to execute that choreography.

Correlation: The concept of correlation describes how conversations are structured and which properties must be exchanged to retain the semantic consistency of the conversation.

Event: The occurrence of an event that can be handled by a WSCI event handler. WSCI supports the following event types: receipt of a message, occurrence of a timeout and occurrence of a fault.

Exception: Declares the exceptional behavior that may be exhibited by a Service at a given point in a choreography.

Execution Context: describes the environment in which sets of activities are performed. It contains the declarations of the context pertaining to the set of activities, plus, recursively, the declarations inherited from the execution context of the activity containing this set of activities.

Fault: An event that expresses a failure in performing an action or activity. Event handlers declared in the same or parent contexts may catch the fault event.

Global Model: Describes a multi-participant view of the overall message exchange as a collection of links between the operations of communicating services.

Interface: Describes the observable behavior of a service as it participates in a message Exchange.

Message Correlation: A mechanism by which a message received by a service is associated with a particular conversation. See also Correlation.

Nested Process: A process that is local to a context. A nested process is defined within the context of complex activities and can be referenced only from within the context defining it.

Operation: An abstract description of an atomic message exchange supported by the service. In the current specification, the Operation maps to WSDL operations.

Process: A process defines one activity set that is performed exactly once, and all activities in that activity set are performed in sequential order. Referencing it from other processes or activities can reuse the behavior described by a process.

Process Instance: A process being executed.

Role: The identification of some behavior within a process definition. The `role` attribute associates a role name with WSCI actions that represent that behavior.

Service: An instance of a Service Type, i.e. the actual implementation of a Service exhibiting the behavior described for the service type. See also Web Service below.

Service type: Identifies a class of Web services that exhibit the same behavior with respect to a given message exchange.

Thread: A portion of a program that can run independently of and concurrently with other portions of the program.

Top-level process: These processes are defined at the interface level and can be referenced from everywhere within the interface.

Transaction: Transactions are used to model the behavior of a service asserting that a certain number of activities should be treated as a single unit of work; a service uses a transaction to communicate to other services its ability to either completely execute those activity or to restore the consistent state prior to the execution.

Transaction Instance: A transaction being executed.

Web Service: Web services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web service is an autonomous, well-defined, standards-based component that can be accessed via established

Web-based protocols.

8.4 WSCI Schema

Note: The following WSCI schemas have been validated using today's popular XML tools.

8.4.1 Standard

```
<xsd:schema targetNamespace="http://www.w3.org/2002/07/wsci10"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:wsci="http://www.w3.org/2002/07/wsci10"
            elementFormDefault="qualified"
            blockDefault="#all">

<!-- Simple type definitions -->

<xsd:simpleType name="listOfQName">
  <xsd:annotation>
    <xsd:documentation>
      Simple type representing a list of QNames.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:list itemType="xsd:QName"/>
</xsd:simpleType>

<xsd:simpleType name="transactionType">
  <xsd:annotation>
    <xsd:documentation>
      Allowed transaction types.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="atomic"/>
    <xsd:enumeration value="open"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="instantiationType">
  <xsd:annotation>
    <xsd:documentation>
      Allowed instantiation types: message or other.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="message"/>
    <xsd:enumeration value="other"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="timeConstraintType">
```

```
  <xsd:annotation>
    <xsd:documentation>
      Time constraint type: duration or instance.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="duration"/>
    <xsd:enumeration value="dateTime"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="opName">
  <xsd:annotation>
    <xsd:documentation>
      Simple type representing a WSDL operation name
      (QName/NCName).
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[\i-[:]][\c-[:]]*:[\i-[:]][\c-[:]]*/[\i-[:]][\c-
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="opNameList">
  <xsd:annotation>
    <xsd:documentation>
      Simple type representing a list of WSDL
      operation names.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:list itemType="wsci:opName"/>
</xsd:simpleType>

<xsd:simpleType name="twoOpNames">
  <xsd:annotation>
    <xsd:documentation>
      Simple type representing two WSDL operation names.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="wsci:opNameList">
    <xsd:length value="2"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="expression">
  <xsd:annotation>
    <xsd:documentation>
      Simple type representing an expression,
      possibly but not necessarily XPath.
      Cannot hold an empty string.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:simpleType name="timeReference">
  <xsd:annotation>
    <xsd:documentation>
      Time reference: a property name or an activity name
      plus the @start or @end designator.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[\i-[:]][\c-[:]]*:[\i-[:]][\c-[:]]*(@start|@end)
  </xsd:restriction>
</xsd:simpleType>

<!-- Complex type and element definitions (generic) -->

<xsd:complexType name="documented">
  <xsd:annotation>
    <xsd:documentation>
       This type is extended by all elements that
       allow documentation.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
  <xsd:element ref="wsci:documentation" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="documentation">
  <xsd:annotation>
    <xsd:documentation>
       This element allows documentation to appear as
       mixed content using any schema.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:any namespace="##other" processContents="skip"
               minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="selector" type="wsci:selector"/>

<xsd:complexType name="selector">
  <xsd:annotation>
    <xsd:documentation>
      A property selector. Defines how a property value is
      instantiated from a message part given the type
      definition of the message part and an optional XPath
      expression. The element and type attributes are mutually
      exclusive. At least one type reference must be used.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:documented">
```

```
      <xsd:sequence>
        <xsd:any namespace="##other" processContents="strict"
                 minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="property" type="xsd:QName"
                     use="required"/>
      <xsd:attribute name="element" type="xsd:QName"
                     use="optional"/>
      <xsd:attribute name="type" type="xsd:QName"
                     use="optional"/>
      <xsd:attribute name="xpath" type="wsci:expression"
                     use="optional"/>
      <xsd:anyAttribute namespace="##other"
                        processContents="strict"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="correlation" type="wsci:correlation"/>

<xsd:complexType name="correlation">
  <xsd:annotation>
    <xsd:documentation>
      A correlation definition. Names a correlation and
      specifies the properties that are used to identify the
      correlation instance.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:documented">
      <xsd:attribute name="name" type="xsd:NCName"
                     use="required"/>
      <xsd:attribute name="property" type="wsci:listOfQName"
                     use="required"/>
      <xsd:attribute name="extends" type="xsd:QName"
                     use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="correlate">
  <xsd:annotation>
    <xsd:documentation>
      Correlates an action by referencing the correlation
      definition and indicating whether that action is part
      of the correlation instantiation or part of an existing
      correlation instance.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="correlation" type="xsd:QName"
                 use="required"/>
  <xsd:attribute name="instantiation" type="xsd:boolean"
                 use="optional" default="false"/>
</xsd:complexType>

<!-- Generic type definitions used by activity types -->
```

```
<xsd:complexType name="activity">
  <xsd:annotation>
    <xsd:documentation>
       The base type for all activity elements. It defines
       the optional name attribute and includes the
       documentation element in the contents.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:documented">
      <xsd:attribute name="name" type="xsd:NCName"
                     use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="activity" abstract="true" type="wsci:activity"/>

<xsd:group name="activitySet">
  <xsd:annotation>
    <xsd:documentation>
      A set of activities that perform in the same context.
      Allows the optional context element, and zero or more
      activities from all the activity types.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="context" type="wsci:context"
                 minOccurs="0" maxOccurs="1">
      <xsd:key name="contextProcessConstraint">
        <xsd:selector xpath="./wsci:process"/>
        <xsd:field xpath="@name"/>
      </xsd:key>
      <xsd:key name="contextPropertyConstraint">
        <xsd:selector xpath="./wsci:property"/>
        <xsd:field xpath="@name"/>
      </xsd:key>
    </xsd:element>
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="action" type="wsci:action"/>
      <xsd:element name="all" type="wsci:all"/>
      <xsd:element name="call" type="wsci:call"/>
      <xsd:element name="choice" type="wsci:choice">
        <xsd:unique name="choiceFaultConstraint">
          <xsd:selector xpath="./wsci:onFault"/>
          <xsd:field xpath="@code"/>
        </xsd:unique>
        <xsd:key name="choiceTimeoutConstraint">
          <xsd:selector xpath="./wsci:onTimeout"/>
          <xsd:field xpath="@property"/>
        </xsd:key>
      </xsd:element>
      <xsd:element name="compensate"
                   type="wsci:compensate"/>
      <xsd:element name="delay" type="wsci:delay"/>
```

```
        <xsd:element name="empty" type="wsci:empty"/>
        <xsd:element name="fault" type="wsci:fault"/>
        <xsd:element name="foreach" type="wsci:foreach"/>
        <xsd:element name="join" type="wsci:join"/>
        <xsd:element name="sequence" type="wsci:sequence"/>
        <xsd:element name="spawn" type="wsci:spawn"/>
        <xsd:element name="switch" type="wsci:switch">
          <xsd:key name="switchConditionConstraint">
            <xsd:selector xpath="./wsci:case"/>
            <xsd:field xpath="./wsci:condition"/>
          </xsd:key>
        </xsd:element>
        <xsd:element name="until" type="wsci:until"/>
        <xsd:element name="while" type="wsci:while"/>
        <xsd:element ref="wsci:activity"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:group>

  <xsd:complexType name="condition">
    <xsd:annotation>
      <xsd:documentation>
         A condition expression. The expression is given
         as character data, the common understanding is
         that of an XPath expression. Extension attributes
         can be used to express other semantics.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:extension base="wsci:expression">
        <xsd:anyAttribute namespace="##other"
                          processContents="strict"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="eventHandler">
    <xsd:annotation>
      <xsd:documentation>
         The base type for all activity elements. It defines
         the optional name attribute and includes the
         documentation element in the contents.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:documented"/>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="onMessage">
    <xsd:annotation>
      <xsd:documentation>
        An event handler that is triggered by an
        inbound message. The first action activity
        that precedes the activity set must complete
        for this activity set to perform.
```

```
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:eventHandler">
        <xsd:sequence>
          <xsd:element name="action" type="wsci:action"/>
          <xsd:group ref="wsci:activitySet"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="onTimeout">
    <xsd:annotation>
      <xsd:documentation>
        An event handler that is triggered by a timeout.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:eventHandler">
        <xsd:group ref="wsci:activitySet"/>
        <xsd:attribute name="property" type="xsd:QName"
                       use="required"/>
        <xsd:attribute name="type" type="wsci:timeConstraintType"
                       use="optional" default="duration"/>
        <xsd:attribute name="reference" type="wsci:timeReference"
                       use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="onFault">
    <xsd:annotation>
      <xsd:documentation>
        An event handler that is triggered by a fault.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:eventHandler">
        <xsd:group ref="wsci:activitySet"/>
        <xsd:attribute name="code" type="xsd:QName"
                       use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:group name="anyEvent">
    <xsd:annotation>
      <xsd:documentation>
        An event handler: onMessage, onTimeout or onFault.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice>
      <xsd:element name="onMessage" type="wsci:onMessage"/>
      <xsd:element name="onTimeout" type="wsci:onTimeout"/>
      <xsd:element name="onFault" type="wsci:onFault"/>
```

```
    </xsd:choice>
  </xsd:group>

  <!-- Type definitions for activities -->

  <xsd:complexType name="action">
    <xsd:annotation>
      <xsd:documentation>
        The action activity references an abstract operation
        definition against which the action will occur.
        It also identifies which correlations are used to
        associate a message instance with an activity context.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:activity">
        <xsd:sequence>
          <xsd:element name="correlate" type="wsci:correlate"
                       minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="call" type="wsci:call"
                       minOccurs="0" maxOccurs="1"/>
          <xsd:any namespace="##other" processContents="strict"
                   minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="operation" type="wsci:opName"
                       use="optional"/>
        <xsd:attribute name="role" type="xsd:QName"
                       use="optional"/>
        <xsd:anyAttribute namespace="##other"
                          processContents="strict"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="all">
    <xsd:annotation>
      <xsd:documentation>
        Performs all the activities in any order.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:activity">
        <xsd:group ref="wsci:activitySet"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="call">
    <xsd:annotation>
      <xsd:documentation>
        Invokes a process and waits for it complete.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:activity">
        <xsd:sequence/>
```

```
      <xsd:attribute name="process" type="xsd:NCName"
                     use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="choice">
  <xsd:annotation>
    <xsd:documentation>
      Selects one activity set and performs it.
      The activity set is selected by an event
      handler or time onMessage, onTimeout or
      onFault. At least two event handlers must
      be specified.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:group ref="wsci:anyEvent"
                 minOccurs="2" maxOccurs="unbounded"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="compensate">
  <xsd:annotation>
    <xsd:documentation>
      Compensate for all completed instances of the
      named transaction that has not been compensated
      for yet.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:sequence/>
      <xsd:attribute name="transaction" type="xsd:NCName"
                     use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="delay">
  <xsd:annotation>
    <xsd:documentation>
      Represents the passage of time.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:attribute name="property" type="xsd:QName"
                     use="required"/>
      <xsd:attribute name="type" type="wsci:timeConstraintType"
                     use="optional" default="duration"/>
      <xsd:attribute name="reference" type="wsci:timeReference"
                     use="optional"/>
    </xsd:extension>
```

```
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="empty">
  <xsd:annotation>
    <xsd:documentation>
      An activity that does nothing.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity"/>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="fault">
  <xsd:annotation>
    <xsd:documentation>
      Signals a fault in the process and branches into
      exception handling.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:attribute name="code" type="xsd:QName"
                     use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="foreach">
  <xsd:annotation>
    <xsd:documentation>
      Repeats the activity set for each item in
      the pre-selected item list.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:group ref="wsci:activitySet"/>
      <xsd:attribute name="select" type="wsci:expression"
                     use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="join">
  <xsd:annotation>
    <xsd:documentation>
      Waits for all nested process instances to complete.
    </xsd:documentation>  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:sequence/>
      <xsd:attribute name="process" type="xsd:NCName"
                     use="required"/>
    </xsd:extension>
```

```
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="sequence">
  <xsd:annotation>
    <xsd:documentation>
      Performs all the activities in sequence.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:group ref="wsci:activitySet"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="spawn">
  <xsd:annotation>
    <xsd:documentation>
      Instantiates a process without waiting for
      it to complete.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:sequence/>
      <xsd:attribute name="process" type="xsd:NCName"
                     use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="switch">
  <xsd:annotation>
    <xsd:documentation>
      Selects one activity set and performs it.
      The condition of each case are evaluated in
      order and the first condition to be met (evaluate
      to true) will cause the activity set of that case
      to be performed and the switch activity to complete.
      If no condition is met, the activity set of the
      default case is performed and the switch activity
      completes. If the default case is missing, the
      switch activity completes immediately.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:activity">
      <xsd:sequence>
        <xsd:element name="case"
                     minOccurs="1" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="wsci:documented">
                <xsd:sequence>
                  <xsd:element name="condition"
```

```
                                    type="wsci:condition"/>
                    <xsd:group ref="wsci:activitySet"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="default"
                       minOccurs="0" maxOccurs="1">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="wsci:documented">
                  <xsd:group ref="wsci:activitySet"/>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="until">
    <xsd:annotation>
      <xsd:documentation>
        Repeats the activity set until the condition is
        not met (evaluates to false). The activity set
        will be repeated one or more times.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:activity">
        <xsd:sequence>
          <xsd:element name="condition"
                       type="wsci:condition"/>
          <xsd:group ref="wsci:activitySet"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="while">
    <xsd:annotation>
      <xsd:documentation>
        Repeats the activity set while the condition is
        met (evaluates to true). The activity set will be
        repeated zero or more times.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:activity">
        <xsd:sequence>
          <xsd:element name="condition"
                       type="wsci:condition"/>
          <xsd:group ref="wsci:activitySet"/>
        </xsd:sequence>
```

```
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>


<!-- Type definitions for activity context -->

<xsd:complexType name="context">
  <xsd:annotation>
    <xsd:documentation>
      Defines the context for an activity set.
      All activities within the set perform in this context.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="process" type="wsci:process"/>
      <xsd:element name="property" type="wsci:property"/>
    </xsd:choice>
    <xsd:element name="exception"
                 minOccurs="0" maxOccurs="1">
      <xsd:annotation>
        <xsd:documentation>
          Defines exception handling.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:group ref="wsci:anyEvent"
                   minOccurs="1" maxOccurs="unbounded"/>
      </xsd:complexType>
      <xsd:unique name="exceptionFaultConstraint">
        <xsd:selector xpath="./wsci:onFault"/>
        <xsd:field xpath="@code"/>
      </xsd:unique>
      <xsd:key name="exceptionTimeoutConstraint">
        <xsd:selector xpath="./wsci:onTimeout"/>
        <xsd:field xpath="@property"/>
      </xsd:key>
    </xsd:element>
    <xsd:element name="transaction"
                 minOccurs="0" maxOccurs="1">
      <xsd:annotation>
        <xsd:documentation>
          Defines the name of a transaction and the transaction
          type. Optionally defines compensating activity.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="compensation"
                       minOccurs="0" maxOccurs="1">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="wsci:documented">
                  <xsd:group ref="wsci:activitySet"/>
                </xsd:extension>
```

```
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName"
                     use="required"/>
      <xsd:attribute name="type" type="wsci:transactionType"
                     use="optional" default="atomic"/>
      <xsd:attribute name="retries" type="xsd:QName"
                     use="optional"/>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="property">
  <xsd:annotation>
    <xsd:documentation>
      Defines a property within a context.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:documented">
      <xsd:sequence>
        <xsd:element name="value"
                     minOccurs="0" maxOccurs="1">
          <xsd:complexType mixed="true">
            <xsd:sequence>
              <xsd:any namespace="##other" processContents="skip"
                       minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:QName"
                     use="required"/>
      <xsd:attribute name="select" type="wsci:expression"
                     use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- Element definitions -->

<xsd:complexType name="process">
  <xsd:annotation>
    <xsd:documentation>
      Defines a process. A process is an activity set that
      is not contained within any other activity.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci:nameRequired">
      <xsd:group ref="wsci:activitySet"/>
      <xsd:attribute name="instantiation" type="wsci:instantiationType"
                     use="optional" default="message"/>
```

```
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="nameRequired">
  <xsd:complexContent>
    <xsd:restriction base="wsci:activity">
      <xsd:sequence>
        <xsd:element ref="wsci:documentation" minOccurs="0" maxOccurs="1
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName"
                     use="required"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="interface">
  <xsd:annotation>
    <xsd:documentation>
      TBD.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="wsci:documented">
        <xsd:sequence>
          <xsd:element name="process" type="wsci:process"
                       minOccurs="1" maxOccurs="unbounded">
            <xsd:unique name="processContextNameConstraint">
              <xsd:selector xpath="./*|./wsci:context/*"/>
              <xsd:field xpath="@name"/>
            </xsd:unique>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:NCName"
                       use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:key name="processNameConstraint">
    <xsd:selector xpath="./wsci:process"/>
    <xsd:field xpath="@name"/>
  </xsd:key>
</xsd:element>

<xsd:element name="model">
  <xsd:annotation>
    <xsd:documentation>
      TBD.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="wsci:documented">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
```

```
            <xsd:element name="interface">
              <xsd:complexType>
                <xsd:attribute name="ref" type="xsd:QName"
                               use="required"/>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="connect" type="wsci:connect"/>
          </xsd:sequence>
          <xsd:attribute name="name" type="xsd:NCName"
                         use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:key name="interfaceNameConstraint">
      <xsd:selector xpath="./wsci:interface"/>
      <xsd:field xpath="@name"/>
    </xsd:key>
  </xsd:element>

  <xsd:complexType name="connect">
    <xsd:annotation>
      <xsd:documentation>
        Connects two operations in opposing port types.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wsci:documented">
        <xsd:sequence>
          <xsd:any namespace="##other" processContents="strict"
                   minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="operation" type="wsci:twoOpNames"
                       use="optional"/>
        <xsd:anyAttribute namespace="##other"
                          processContents="strict"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  </xsd:schema>
```

8.4.2 Extended: Locate

```
<xsd:schema targetNamespace="http://www.w3.org/2002/07/wsci10/locate"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:wsci-core="http://www.w3.org/2002/07/wsci10"
            xmlns:wsci-locate="http://www.w3.org/2002/07/wsci10/locate"
            elementFormDefault="qualified"
            blockDefault="#all">

<xsd:import namespace="http://www.w3.org/2002/07/wsci10"/>

<xsd:element name="locator" type="wsci-locate:locator"/>
```

```
<xsd:complexType name="locator">
  <xsd:annotation>
    <xsd:documentation>
      Defines a locator. An extension element and optional
      extension attributes are used to define the mechanism
      by which the service instance is identified.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="wsci-core:documented">
      <xsd:sequence>
        <xsd:any namespace="##other" processContents="strict"
                 minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName"
                     use="required"/>
      <xsd:anyAttribute namespace="##other"
                        processContents="strict"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="locate" type="wsci-locate:locate"/>

<xsd:complexType name="locate">
  <xsd:annotation>
    <xsd:documentation>
      Locates a service instance. The property attribute
      names the properties that are used to locate the
      service instance. The locator attribute references
      a locator definition. If absent, the service is located
      by its end-point URI given by a single property.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="property" type="wsci-core:listOfQName"
                 use="optional"/>
  <xsd:attribute name="locator" type="xsd:QName"
                 use="optional"/>
</xsd:complexType>

</xsd:schema>
```